[COM6513] Assignment 2: Topic Classification with a Feedforward Network

Instructor: Nikos Aletras

The goal of this assignment is to develop a Feedforward neural network for topic classification.

For that purpose, you will implement:

- Text processing methods for transforming raw text data into input vectors for your network (**1 mark**)
- A Feedforward network consisting of:
  - **One-hot** input layer mapping words into an **Embedding weight matrix** (**1 mark**)
  - **One hidden layer** computing the mean embedding vector of all words in input followed by a **ReLU activation function** (**1 mark**)
  - **Output layer** with a **softmax** activation. (**1 mark**)
- The Stochastic Gradient Descent (SGD) algorithm with **back-propagation** to learn the weights of your Neural network. Your algorithm should:
  - Use (and minimise) the **Categorical Cross-entropy loss** function (**1 mark**)
  - Perform a **Forward pass** to compute intermediate outputs (**3 marks**)
  - Perform a **Backward pass** to compute gradients and update all sets of weights (**6 marks**)
  - Implement and use **Dropout** after each hidden layer for regularisation (**2 marks**)
- Discuss how did you choose hyperparameters? You can tune the learning rate (hint: choose small values), embedding size {e.g. 50, 300, 500}, the dropout rate {e.g. 0.2, 0.5} and the learning rate. Please use tables or graphs to show training and validation performance for each hyperparameter combination (**2 marks**).
- After training a model, plot the learning process (i.e. training and validation loss in each epoch) using a line plot and report accuracy. Does your model overfit, underfit or is about right? (**1 mark**).
- Re-train your network by using pre-trained embeddings (GloVe (https://nlp.stanford.edu/projects/glove/)) trained on large corpora. Instead of randomly initialising the embedding weights matrix, you should initialise it with the pre-trained weights. During training, you should not update them (i.e. weight freezing) and backprop should stop before computing gradients for updating embedding weights. Report results by performing hyperparameter tuning and plotting the learning process. Do you get better performance? (**3 marks**).
- Extend you Feedforward network by adding more hidden layers (e.g. one more or two). How does it affect the performance? Note: You need to repeat hyperparameter tuning, but the number of combinations grows exponentially. Therefore, you need to choose a subset of all possible combinations (**4 marks**)
- Provide well documented and commented code describing all of your choices. In general, you are free to make decisions about text processing (e.g. punctuation, numbers, vocabulary size) and hyperparameter values. We expect to see justifications and discussion for all of your choices (**2 marks**).
- Provide efficient solutions by using Numpy arrays when possible. Executing the whole notebook with your code should not take more than 10 minutes on any standard computer (e.g. Intel Core i5 CPU, 8 or 16GB RAM) excluding hyperparameter tuning runs and loading the pretrained vectors. You can find tips in Lab 1 (**2 marks**).

Data

The data you will use for the task is a subset of the AG News Corpus (http://groups.di.unipi.it/~gulli/AG_corpus_of_news_articles.html) and you can find it in the `./data_topic` folder in CSV format:

- `data_topic/train.csv` : contains 2,400 news articles, 800 for each class to be used for training.
- `data_topic/dev.csv` : contains 150 news articles, 50 for each class to be used for hyperparameter selection and monitoring the training process.
- `data_topic/test.csv` : contains 900 news articles, 300 for each class to be used for testing.

Pre-trained Embeddings

You can download pre-trained GloVe embeddings trained on Common Crawl (840B tokens, 2.2M vocab, cased, 300d vectors, 2.03 GB download) from here (http://nlp.stanford.edu/data/glove.840B.300d.zip). No need to unzip, the file is large.

Save Memory

To save RAM, when you finish each experiment you can delete the weights of your network using `del W` followed by Python's garbage collector `gc.collect()`

Submission Instructions

You should submit a Jupyter Notebook file (assignment2.ipynb) and an exported PDF version (you can do it from Jupyter: `File->Download as->PDF via Latex`).

You are advised to follow the code structure given in this notebook by completing all given funtions. You can also write any auxilliary/helper functions (and arguments for the functions) that you might need but note that you can provide a full solution without any such functions. Similarly, you can just use only the packages imported below but you are free to use any functionality from the Python Standard Library (https://docs.python.org/3/library/index.html), NumPy, SciPy (excluding built-in softmax funtcions) and Pandas. You are **not allowed to use any third-party library** such as Scikit-learn (apart from metric functions already provided), NLTK, Spacy, Keras, Pytorch etc.. You should mention if you've used Windows to write and test your code because we mostly use Unix based machines for marking (e.g. Ubuntu, MacOS).

There is no single correct answer on what your accuracy should be, but correct implementations usually achieve F1-scores around 80% or higher. The quality of the analysis of the results is as important as the accuracy itself.

This assignment will be marked out of 30. It is worth 30% of your final grade in the module.

The deadline for this assignment is **23:59 on Mon, 9 May 2022** and it needs to be submitted via Blackboard. Standard departmental penalties for lateness will be applied. We use a range of strategies to **detect unfair means (https://www.sheffield.ac.uk/ssid/unfair-means/index)**, including Turnitin which helps detect plagiarism. Use of unfair means would result in getting a failing grade.

In [1]:

```
import pandas as pd
import numpy as np
from collections import Counter
import re
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import random
from time import localtime, strftime
from scipy.stats import spearmanr,pearsonr
import zipfile
import gc

# fixing random seed for reproducibility
random.seed(123)
np.random.seed(123)
```

Transform Raw texts into training and development data

First, you need to load the training, development and test sets from their corresponding CSV files (tip: you can use Pandas dataframes).

In [2]:

```
topic_dev = pd.read_csv('dev.csv')        # Reading CSV file
topic_dev.columns=['Label','Text']        # Column names are given here
topic_test = pd.read_csv('test.csv')
topic_test.columns = ['Label','Text']
topic_train = pd.read_csv('train.csv')
topic_train.columns = ['Label','Text']
```

In [3]:

```
topic_dev_texts = list(topic_dev['Text'])        # Seprating text data
topic_train_texts = list(topic_train['Text'])
topic_test_texts = list(topic_test['Text'])
```

In [4]:

```
topic_dev.head()
```

Out[4]:

	Label	Text
0	1	Parts of Los Angeles international airport are...
1	1	AFP - Facing a issue that once tripped up his ...
2	1	The leader of militant Lebanese group Hezbolla...
3	1	JAKARTA : ASEAN finance ministers ended a meet...
4	1	The death toll in the Russian schoolhouse sieg...

Create input representations

To train your Feedforward network, you first need to obtain input representations given a vocabulary. One-hot encoding requires large memory capacity. Therefore, we will instead represent documents as lists of vocabulary indices (each word corresponds to a vocabulary index).

Text Pre-Processing Pipeline

To obtain a vocabulary of words. You should:

- tokenise all texts into a list of unigrams (tip: you can re-use the functions from Assignment 1)
- remove stop words (using the one provided or one of your preference)
- remove unigrams appearing in less than K documents
- use the remaining to create a vocabulary of the top-N most frequent unigrams in the entire corpus.

In [5]:

```
stop_words = ['a', 'ad', 'after', 'again', 'all', 'also', 'am', 'an', 'and', 'any',
    'are', 'as', 'at', 'be', 'because', 'been', 'being', 'between', 'both',
    'but', 'by', 'can', 'could', 'does', 'each', 'ed', 'eg', 'either', 'etc',
    'even', 'ever', 'every', 'for', 'from', 'had', 'has', 'have', 'he', 'her',
    'hers', 'herself', 'him', 'himself', 'his', 'i', 'ie', 'if', 'in', 'inc',
    'into', 'is', 'it', 'its', 'itself', 'li', 'll', 'ltd', 'may', 'maybe',
    'me', 'might', 'mine', 'minute', 'minutes', 'must', 'my', 'myself',
    'neither', 'nor', 'now', 'of', 'on', 'only', 'or', 'other', 'our', 'ours',
    'ourselves', 'own', 'same', 'seem', 'seemed', 'shall', 'she', 'some',
    'somehow', 'something', 'sometimes', 'somewhat', 'somewhere', 'spoiler',
    'spoilers', 'such', 'suppose', 'that', 'the', 'their', 'theirs', 'them',
    'themselves', 'there', 'these', 'they', 'this', 'those', 'thus', 'to',
    'today', 'tomorrow', 'us', 've', 'vs', 'was', 'we', 'were', 'what',
    'whatever', 'when', 'whenever', 'where', 'whereby', 'which', 'who', 'whom',
    'whose', 'will', 'with', 'yesterday', 'you', 'your', 'yours', 'yourself',
    'yourselves']
```

Unigram extraction from a document

You first need to implement the `extract_ngrams` function. It takes as input:

- `x_raw` : a string corresponding to the raw text of a document
- `ngram_range` : a tuple of two integers denoting the type of ngrams you want to extract, e.g. (1,2) denotes extracting unigrams and bigrams.
- `token_pattern` : a string to be used within a regular expression to extract all tokens. Note that data is already tokenised so you could opt for a simple white space tokenisation.
- `stop_words` : a list of stop words
- `vocab` : a given vocabulary. It should be used to extract specific features.

and returns:

- a list of all extracted features.

In [6]:

```
def extract_ngrams(x_raw, ngram_range=(1,3), token_pattern=r'\b[A-Za-z][A-Za-z]+\b', stop_w

    tokenRE = re.compile(token_pattern)

    # first extract all unigrams by tokenising
    x_uni = [w for w in tokenRE.findall(str(x_raw).lower(),) if w not in stop_words]

    # this is to store the ngrams to be returned
    x = []

    if ngram_range[0]==1:
        x = x_uni

    # generate n-grams from the available unigrams x_uni
    ngrams = []
    for n in range(ngram_range[0], ngram_range[1]+1):

        # ignore unigrams
        if n==1: continue

        # pass a list of lists as an argument for zip
        arg_list = [x_uni]+[x_uni[i:] for i in range(1, n)]

        # extract tuples of n-grams using zip
        # for bigram this should look: list(zip(x_uni, x_uni[1:]))
        # align each item x[i] in x_uni with the next one x[i+1].
        # Note that x_uni and x_uni[1:] have different lengths
        # but zip ignores redundant elements at the end of the second list
        # Alternatively, this could be done with for loops
        x_ngram = list(zip(*arg_list))
        ngrams.append(x_ngram)


    for n in ngrams:
        for t in n:
            x.append(t)

    if len(vocab)>0:
        x = [w for w in x if w in vocab]

    return x
```

Create a vocabulary of n-grams

Then the `get_vocab` function will be used to (1) create a vocabulary of ngrams; (2) count the document frequencies of ngrams; (3) their raw frequency. It takes as input:

- `X_raw` : a list of strings each corresponding to the raw text of a document
- `ngram_range` : a tuple of two integers denoting the type of ngrams you want to extract, e.g. (1,2) denotes extracting unigrams and bigrams.
- `token_pattern` : a string to be used within a regular expression to extract all tokens. Note that data is already tokenised so you could opt for a simple white space tokenisation.
- `stop_words` : a list of stop words
- `min_df` : keep ngrams with a minimum document frequency.
- `keep_topN` : keep top-N more frequent ngrams.

and returns:

- `vocab` : a set of the n-grams that will be used as features.
- `df` : a Counter (or dict) that contains ngrams as keys and their corresponding document frequency as values.
- `ngram_counts` : counts of each ngram in vocab

In [7]:

```
def get_vocab(X_raw,
              ngram_range=(1, 3),
              token_pattern=r'\b[A-Za-z]{2,}\b',
              min_df=1,
              keep_topN=0,
              stop_words=stop_words):

    doc_freq = Counter()
    ngram_count = Counter()
    z=[]
    for text in X_raw:
        # A list of ngrams for the given document `text`
        ngram_list = extract_ngrams(text, ngram_range, token_pattern, stop_words)
        doc_freq.update(set(ngram_list))                          # Here we are counting docu
        for ngram in ngram_list:                                  # Here we are counting ngra
            list=[]
            if doc_freq[ngram]>=min_df:
                list.append(ngram)
            ngram_count.update(list)
    vocab = {ngram for ngram, _ in ngram_count.most_common(keep_topN)}   # Here we are extr
    return vocab, doc_freq, ngram_count
```

Now you should use `get_vocab` to create your vocabulary and get document and raw frequencies of unigrams:

In [8]:

```
dev_vocab = get_vocab(topic_dev_texts, ngram_range=(1, 1), keep_topN=5000)[0:1]
train_vocab = get_vocab(topic_train_texts, ngram_range=(1, 1), keep_topN=5000)[0:1]
test_vocab = get_vocab(topic_test_texts, ngram_range=(1, 1), keep_topN=5000)[0:1]

# Set order is random for every new run. Use `sorted` for reproducibility
vocab = sorted(dev_vocab[0].union(train_vocab[0]).union(test_vocab[0]))
```

In [9]:

```
vocab
```

Out[9]:

```
['aaa',
 'aaron',
 'ab',
 'abandon',
 'abandoned',
 'abbey',
 'abby',
 'abducted',
 'abdullah',
 'aber',
 'ability',
 'abkhazia',
 'able',
 'aboard',
 'about',
 'above',
 'abraham',
 'abroad',
```

Then, you need to create vocabulary id -> word and word -> vocabulary id dictionaries for reference:

In [10]:

```
vocab_id_to_word = dict(enumerate(vocab))
word_to_vocab_id = {v: k for k, v in vocab_id_to_word.items()}
```

Convert the list of unigrams into a list of vocabulary indices

Storing actual one-hot vectors into memory for all words in the entire data set is prohibitive. Instead, we will store word indices in the vocabulary and look-up the weight matrix. This is equivalent of doing a dot product between an one-hot vector and the weight matrix.

First, represent documents in train, dev and test sets as lists of words in the vocabulary:

In [11]:

```
def generator_dev ():
    return ([extract_ngrams(doc, ngram_range=(1,1))for doc in topic_dev_texts])
def generator_train ():
    return ([extract_ngrams(doc, ngram_range=(1,1),vocab=vocab)for doc in topic_train_texts
def generator_test ():
    return ([extract_ngrams(doc, ngram_range=(1,1),vocab=vocab)for doc in topic_test_texts]


topic_dev_uni = generator_dev()
topic_train_uni = generator_train()
topic_test_uni = generator_test()
```

topic_dev_ids = [[word_to_vocab_id[uni] for uni in unigrams] for unigrams in topic_dev_uni]

topic_train_ids = [[word_to_vocab_id[uni] for uni in unigrams] for unigrams in topic_train_uni]

topic_test_ids = [[word_to_vocab_id[uni] for uni in unigrams] for unigrams in topic_test_uni]

Then convert them into lists of indices in the vocabulary:

In [12]:

```
def generator_dev_data ():
    return ([[word_to_vocab_id[uni] for uni in unigrams]
                for unigrams in topic_dev_uni])

def generator_train_data ():
    return ([[word_to_vocab_id[uni] for uni in unigrams]
                for unigrams in topic_train_uni])

def generator_test_data ():
    return ([[word_to_vocab_id[uni] for uni in unigrams]
                for unigrams in topic_test_uni])

topic_dev_ids = generator_dev_data()
topic_train_ids = generator_train_data()
topic_test_ids = generator_test_data()
```

In [13]:

```
topic_dev_ids
```

Out[13]:

```
[[4826, 3975, 249, 3459, 154, 6790, 1251, 2015, 225, 5591, 5971, 822],
 [106,
  2397,
  3520,
  4646,
  7024,
  7167,
  2448,
  5131,
  2802,
  933,
  6912,
  81,
  6610,
  3898,
  868,
  4816,
  2901,
```

Put the labels `Y` for train, dev and test sets into arrays:

In [14]:

```
topic_dev_labels = np.array(topic_dev['Label']) - 1  # Starting position of the Label is 1,
topic_train_labels = np.array(topic_train['Label']) - 1
topic_test_labels = np.array(topic_test['Label']) - 1
```

Network Architecture

Your network should pass each word index into its corresponding embedding by looking-up on the embedding matrix and then compute the first hidden layer $\mathbf{h}_1$:

$$\mathbf{h}_1 = \frac{1}{|x|} \sum_i W_i^e, i \in x$$

where $|x|$ is the number of words in the document and W^e is an embedding matrix $|V| \times d$, $|V|$ is the size of the vocabulary and d the embedding size.

Then $\mathbf{h}_1$ should be passed through a ReLU activation function:

$$\mathbf{a}_1 = relu(\mathbf{h}_1)$$

Finally the hidden layer is passed to the output layer:

$$\mathbf{y} = \text{softmax}(\mathbf{a}_1 W)$$

where W is a matrix $d \times |\mathcal{Y}|$, $|\mathcal{Y}|$ is the number of classes.

During training, $\mathbf{a}_1$ should be multiplied with a dropout mask vector (elementwise) for regularisation before it is passed to the output layer.

You can extend to a deeper architecture by passing a hidden layer to another one:

$$\mathbf{h_i} = \mathbf{a}_{i-1} W_i$$

$$\mathbf{a_i} = relu(\mathbf{h_i})$$

Network Training

First we need to define the parameters of our network by initiliasing the weight matrices. For that purpose, you should implement the `network_weights` function that takes as input:

- `vocab_size` : the size of the vocabulary
- `embedding_dim` : the size of the word embeddings
- `hidden_dim` : a list of the sizes of any subsequent hidden layers. Empty if there are no hidden layers between the average embedding and the output layer
- `num_classes` : the number of the classes for the output layer

and returns:

- `W` : a dictionary mapping from layer index (e.g. 0 for the embedding matrix) to the corresponding weight matrix initialised with small random numbers (hint: use numpy.random.uniform with from -0.1 to 0.1)

Make sure that the dimensionality of each weight matrix is compatible with the previous and next weight matrix, otherwise you won't be able to perform forward and backward passes. Consider also using np.float32 precision to save memory.

In [15]:

```
def network_weights(vocab_size=1000, embedding_dim=300, hidden_dim=[], num_classes=3):
    # fixing random seed for reproducibility
    np.random.seed(123)
    dimensions = [vocab_size, embedding_dim] + hidden_dim + [num_classes]
    W = [np.random.randn(*size).astype(np.float32) * np.sqrt(2 / (size[0])) # Here I have u
         for size in zip(*(dimensions[i:] for i in range(2)))]    #https://machinelearningm
    return W
```

In [16]:

```
W = network_weights(vocab_size=3,embedding_dim=4,hidden_dim=[2], num_classes=2)
```

Then you need to develop a `softmax` function (same as in Assignment 1) to be used in the output layer.

It takes as input `z` (array of real numbers) and returns `sig` (the softmax of `z`)

In [17]:

```
def softmax(z):

    #Compute probability for each class
    # Reference : - https://github.com/motynel75/Vanilla-MLP/blob/master/vanilla_MLP.ipynb

    e_z = np.exp(z)
    sig = e_z / np.sum(e_z, axis=1 if e_z.ndim > 1 else None, keepdims=True)
    return sig
```

Now you need to implement the categorical cross entropy loss by slightly modifying the function from Assignment 1 to depend only on the true label `y` and the class probabilities vector `y_preds`:

In [18]:

```
def categorical_loss(y, y_preds):
    l = -np.log(y_preds[y])
    return l
```

Then, implement the `relu` function to introduce non-linearity after each hidden layer of your network (during the forward pass):

$$relu(z_i) = max(z_i, 0)$$

and the `relu_derivative` function to compute its derivative (used in the backward pass):

relu_derivative(z_i)=0, if z_i<=0, 1 otherwise.

Note that both functions take as input a vector z

Hint use .copy() to avoid in place changes in array z

In [19]:

```
def relu(z):
    a = np.maximum(z, 0)  # relu(zi)=max(zi,0)
    return a

def relu_derivative(z):
    dz = z.copy()         # relu_derivative( zi )=0, if  zi <=0, 1 otherwise.
    dz[dz<=0] = 0
    dz[dz>0] = 1
    return dz
```

During training you should also apply a dropout mask element-wise after the activation function (i.e. vector of ones with a random percentage set to zero). The `dropout_mask` function takes as input:

- `size` : the size of the vector that we want to apply dropout
- `dropout_rate` : the percentage of elements that will be randomly set to zeros

and returns:

- `dropout_vec` : a vector with binary values (0 or 1)

In [20]:

```
def dropout_mask(size, dropout_rate):
    dropout_vec = np.random.choice([0, 1], size=size, p=[dropout_rate, 1-dropout_rate])
    return dropout_vec
```

In [21]:

```
print(dropout_mask(10, 0.2))
print(dropout_mask(10, 0.2))
```

```
[1 1 1 1 0 1 1 1 1 1]
[1 1 1 1 1 0 1 1 1 1]
```

Now you need to implement the `forward_pass` function that passes the input x through the network up to the output layer for computing the probability for each class using the weight matrices in `W` . The ReLU activation function should be applied on each hidden layer.

- `x` : a list of vocabulary indices each corresponding to a word in the document (input)
- `W` : a list of weight matrices connecting each part of the network, e.g. for a network with a hidden and an output layer: W[0] is the weight matrix that connects the input to the first hidden layer, W[1] is the weight matrix that connects the hidden layer to the output layer.
- `dropout_rate` : the dropout rate that is used to generate a random dropout mask vector applied after each hidden layer for regularisation.

and returns:

- `out_vals` : a dictionary of output values from each layer: h (the vector before the activation function), a (the resulting vector after passing h from the activation function), its dropout mask vector; and the prediction vector (probability for each class) from the output layer.

In [22]:

```
def forward_pass(x, W, dropout_rate=0.2):

    out_vals = {}
    h_vecs = []
    a_vecs = []
    dropout_vecs = []

    h_1 = np.mean(W[0][x], axis=0)     # Enbedding Layer or Initial Layer
    h_vecs.append(h_1)


    a_1 = relu(h_1)                    # Here we are applying relu activation function on E
    a_vecs.append(a_1)


    mask_vector = dropout_mask(W[0].shape[1], dropout_rate)
    dropout_vecs.append(mask_vector)   # Here we are applying dropout vector on Embedding L
    a_i = a_1 * dropout_vecs[-1]


    # Loop through each hidden layer
    for weights in W[1:-1]:           # Now are doing same thing for eāch Hidden Layer
        h_i = a_i.dot(weights)
        h_vecs.append(h_i)

        a_i = relu(h_i)
        a_vecs.append(a_i)

        mask_vector = dropout_mask(weights.shape[1], dropout_rate)
        dropout_vecs.append(mask_vector)
        a_i = a_i * dropout_vecs[-1]

    y = softmax(a_i.dot(W[-1]))
    out_vals["h"] = np.array(h_vecs)  # Here we are appending everything we have calculated
    out_vals["a"] = np.array(a_vecs)
    out_vals["dropout"] = np.array(dropout_vecs)
    out_vals["y"] = np.array(y)

    return out_vals
```

The `backward_pass` function computes the gradients and updates the weights for each matrix in the network from the output to the input. It takes as input

- `x` : a list of vocabulary indices each corresponding to a word in the document (input)
- `y` : the true label
- `W` : a list of weight matrices connecting each part of the network, e.g. for a network with a hidden and an output layer: W[0] is the weight matrix that connects the input to the first hidden layer, W[1] is the weight matrix that connects the hidden layer to the output layer.
- `out_vals` : a dictionary of output values from a forward pass.
- `learning_rate` : the learning rate for updating the weights.
- `freeze_emb` : boolean value indicating whether the embedding weights will be updated.

and returns:

- `W` : the updated weights of the network.

Hint: the gradients on the output layer are similar to the multiclass logistic regression.

In [23]:

```
def backward_pass(x, y, W, out_vals, lr=0.001, freeze_emb=False):

    g = out_vals['y'] - (np.arange(len(out_vals['y'])) == y) # Here we are finding out the

    g_on_w = np.outer(g, out_vals['a'][-1] * out_vals['dropout'][-1]).T  # Now we are calcu
    #g_on_w = np.dot((out_vals['a']*out_vals['dropout']).T , g.reshape(1,g.shape[0]))  # He

    g = np.dot(g, W[-1].T) # Here we are taking dot product of gradient calculated in previ

    W[-1] = W[-1] - lr * g_on_w  # Here we are updating weights with respect to gradient on
                                 # weight = weight - learning_rate * error * input
    # Looping through each hidden layer
    for i in range(len(W) - 2, 1, -1):   # Now we are doing similar step for each hidden la
        g = g * relu_derivative(out_vals['h'][i]) # After the embedding Layer we need to us
                                                  # https://machinelearningmastery.com/impl
        g_on_w = np.outer(g, out_vals['a'][i - 1] * out_vals['dropout'][i - 1]).T  # Here w
        #g_on_w = np.dot((out_vals['a'][i - 1] * out_vals['dropout'][i - 1]).T , g.reshape(

        g = np.dot(g, W[i].T) # Here we are propogating gradient with respect to weight in

        W[i] = W[i] - lr * g_on_w # Here we are updating the weight on each Hidden Layer.


    if not freeze_emb:
        g = g * relu_derivative(out_vals['h'][0]) # Here we are writing a condition to chec
        W[0][x] = W[0][x] - lr * g

    return W
```

Finally you need to modify SGD to support back-propagation by using the `forward_pass` and `backward_pass` functions.

The `SGD` function takes as input:

- `X_tr` : array of training data (vectors)
- `Y_tr` : labels of `X_tr`
- `W` : the weights of the network (dictionary)
- `X_dev` : array of development (i.e. validation) data (vectors)
- `Y_dev` : labels of `X_dev`
- `lr` : learning rate
- `dropout` : regularisation strength
- `epochs` : number of full passes over the training data
- `tolerance` : stop training if the difference between the current and previous validation loss is smaller than a threshold
- `freeze_emb` : boolean value indicating whether the embedding weights will be updated (to be used by the backward pass function).
- `print_progress` : flag for printing the training progress (train/validation loss)

and returns:

- `weights` : the weights learned
- `training_loss_history` : an array with the average losses of the whole training set after each epoch
- `validation_loss_history` : an array with the average losses of the whole development set after each epoch

In [24]:

```
def SGD(X_tr, Y_tr, W, X_dev, Y_dev, lr=0.001, dropout=0.2, epochs=5,
        tolerance=0.001, freeze_emb=False, early_stopping=True, print_progress=True):
    np.random.seed(123)                    # Random seed is fixed here so that we can get sa
    training_loss_history = []
    validation_loss_history = []


    def zipper(X_tr, Y_tr):
        size = len(X_tr) if len(X_tr) < len(Y_tr) else len(Y_tr)
        retList = []                          # Create training tuples
        for i in range(size):                 # Adding values from two list simultaneously
            retList.append((X_tr[i], Y_tr[i]))
        return retList

    train_docs = zipper(X_tr, Y_tr)

    for epoch in range(epochs):
        np.random.shuffle(train_docs)         # Shuffling to randomise all values
                                              # Reference
        for x_i, y_i in train_docs:
            W = backward_pass(x_i, y_i, W, forward_pass(x_i, W, dropout), lr, freeze_emb)

        # Monitor training and validation loss
        loss_in_training = np.mean([categorical_loss(y_i, forward_pass(x_i, W, dropout)['y'
                                for x_i, y_i in train_docs])

        def zipper(X_dev, Y_dev):
            size = len(X_dev) if len(X_dev) < len(Y_dev) else len(Y_dev)
            retList = []                          # Create training tuples
            for i in range(size):                 # Adding values from two list simultaneousl
                retList.append((X_dev[i], Y_dev[i]))
            return retList

        dev_docs = zipper(X_dev, Y_dev)
        loss_in_dev = np.mean([categorical_loss(y_i, forward_pass(x_i, W, dropout)['y'])
                               for x_i, y_i in dev_docs])
        # Early stopping              # Reference
                                      # previous validation loss - current validation loss; s
        if epoch > 0 and validation_loss_history[-1] - loss_in_dev < tolerance:
            break
        else:
            training_loss_history.append(loss_in_training)
            validation_loss_history.append(loss_in_dev)

        if print_progress:
            print("Epoch:- ",epoch,"  ","Training loss:- ",loss_in_training,"  ","Validatio

    return W, training_loss_history, validation_loss_history
```

Now you are ready to train and evaluate your neural net. First, you need to define your network using the `network_weights` function followed by SGD with backprop:

In [25]:

```
W = network_weights(vocab_size=len(vocab),embedding_dim=300,
                    hidden_dim=[], num_classes=3)

for i in range(len(W)):
    print('Shape W'+str(i), W[i].shape)

W, training_loss_count, dev_loss_count = SGD(X_tr=topic_train_ids,
                           Y_tr=topic_train_labels,
                           W=W,
                           X_dev=topic_dev_ids,
                           Y_dev=topic_dev_labels,
                           lr=0.15,
                           dropout=0.2,
                           epochs=100)
```

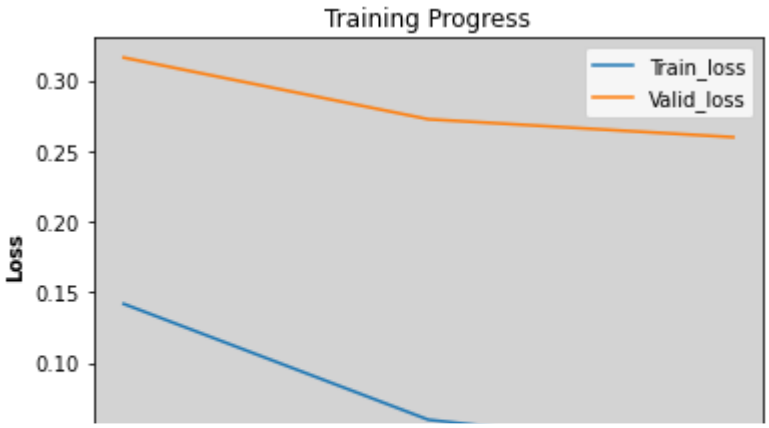
```
Shape W0 (7609, 300)
Shape W1 (300, 3)
Epoch:-  0    Training loss:-  0.14167893629662082    Validation loss:-  0.3
160360111160649
Epoch:-  1    Training loss:-  0.059607705408811974    Validation loss:-  0.
272390480757055
Epoch:-  2    Training loss:-  0.0356398982117245    Validation loss:-  0.25
96920979247858
```

Plot the learning process:

In [26]:

```
plt.plot(training_loss_count, label='Train_loss')
plt.plot(dev_loss_count, label='Valid_loss')

plt.xlabel('Epochs',fontweight='bold')
plt.ylabel('Loss',fontweight='bold')

plt.title('Training Progress')
ax = plt.axes()
ax.set_facecolor("lightgray")


plt.legend()

plt.show()
```

```
<ipython-input-26-dce27189f6e6>:8: MatplotlibDeprecationWarning: Adding an
axes using the same arguments as a previous axes currently reuses the earl
ier instance.  In a future version, a new instance will always be created
and returned.  Meanwhile, this warning can be suppressed, and the future b
ehavior ensured, by passing a unique label to each axes instance.
  ax = plt.axes()
```



Compute accuracy, precision, recall and F1-Score:

In [27]:

```
def zipper(topic_test_ids, topic_test_labels):
    size = len(topic_test_ids) if len(topic_test_ids) < len(topic_test_labels) else len(top
    retList = []                        # Create training tuples
    for i in range(size):               # Adding values from two list simultaneously
        retList.append((topic_test_ids[i], topic_test_labels[i]))
    return retList

test_docs = zipper(topic_test_ids, topic_test_labels)
```

In [28]:

```
preds_te = [np.argmax(forward_pass(x, W, dropout_rate=0.0)['y'])
            for x, y in test_docs]
args = topic_test_labels, preds_te

print('Accuracy:', accuracy_score(*args))
print('Precision:', precision_score(*args, average='macro'))
print('Recall:', recall_score(*args, average='macro'))
print('F1-Score:', f1_score(*args, average='macro'))
```

```
Accuracy: 0.8587319243604005
Precision: 0.8606938846623032
Recall: 0.8587105165366035
F1-Score: 0.8588409627938643
```

Discuss how did you choose model hyperparameters ?

I have used 'He Weight Initialisation'

As a result of initializing the weights with NP.random.uniform(-0.5, 0.5)* and using ReLU as the activation function, when the activation function is passed using the uniform distribution method, half of the layer will output zero. He initialization is similar to Xavier initialization where the number of neurons in the previous layer is given importance. But the factor is multiplied by two instead of one. I have used Embedding size, Learning Rate and Droptout as parameters for Hyperparameter tunning.

Embedding dimension

Larger the Embedding Dimension then Larger will be the Training Time In the case of too large embedding dimensions, there might be an excessive amount of useless information. This would have an adverse effect on performance if the model is overfitting. A small embedding dimension will not provide enough information for representing a word relation.

Fixed parameters: `lr=0.15, tolerance=0.001, dropout=0.2`

Trial	embedding_dim	Epochs	Tr. loss	Val. loss	F1-score
0	300	5	0.2997	0.4866	0.8592
1	400	5	0.2723	0.3679	0.8513
2	500	7	0.1673	0.3869	0.8095
3	800	7	0.1641	0.3573	0.8389
4	1000	9	0.1342	0.2536	0.8581

Learning Rate

A model's learning rate determines how much to change each time the model weights are updated in response to the estimated error. A high learning rate may cause the model to overshoot and lead to divergent behavior (low epochs required) If the learning rate is too small, the model will need a lot of updates to the weights before the loss is converged (epochs required are high).

Fixed parameters: `embedding_dim=300, dropout=0.2, tolerance=0.001`

Trial	lr	Epochs	Tr. loss	Val. loss	F1-score

Trial	lr	Epochs	Tr. loss	Val. loss	F1-score
0	0.1503	5	0.2836	0.4794	0.8562
1	0.16	4	0.3134	0.4680	0.8181
2	0.125	8	0.2389	0.3193	0.8549
3	0.13	7	0.2500	0.4384	0.8470
4	0.1361	5	0.3923	0.4830	0.8452

Dropout Rate (Regularisation)

Dropout rate is used to prevent the model from overfitting. A high dropout rate will result in most layer outputs being ignored, resulting in an under fitted model.

Fixed parameters: `embedding_dim=300, lr=0.15, tolerance=0.001`

Trial	dropout	Epochs	Tr. loss	Val. loss	F1-score
0	0.2	5	0.2993	0.4856	0.8592
1	0.3	7	0.2398	0.3987	0.8120
2	0.4	5	0.4273	0.4645	0.8523
3	0.5	6	0.3573	0.4982	0.7728
4	0.1	4	0.2832	0.4612	0.8233

Use Pre-trained Embeddings

Now re-train the network using GloVe pre-trained embeddings. You need to modify the `backward_pass` function above to stop computing gradients and updating weights of the embedding matrix.

Use the function below to obtain the embedding martix for your vocabulary. Generally, that should work without any problem. If you get errors, you can modify it.

In [29]:

```
def get_glove_embeddings(f_zip, f_txt, word2id, emb_size=300):

    w_emb = np.zeros((len(word2id), emb_size))

    with zipfile.ZipFile(f_zip) as z:
        with z.open(f_txt) as f:
            for line in f:
                line = line.decode('utf-8')
                word = line.split()[0]

                if word in vocab:
                    emb = np.array(line.strip('\n').split()[1:]).astype(np.float32)
                    w_emb[word2id[word]] +=emb
    return w_emb
```

In [30]:

```
w_glove = get_glove_embeddings("glove.840B.300d .zip","glove.840B.300d.txt",word_to_vocab_i
```

First, initialise the weights of your network using the `network_weights` function. Second, replace the weigths of the embedding matrix with `w_glove` . Finally, train the network by freezing the embedding weights:

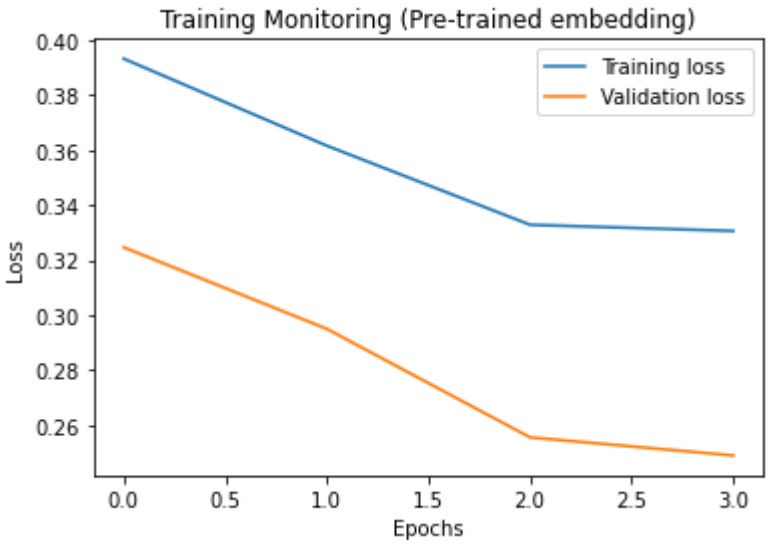
In [31]:

```
W = network_weights(vocab_size=len(vocab),
                    embedding_dim=300,
                    num_classes=3)
W[0] = w_glove
for i in range(len(W)):
    print(f'Shape W{i} {W[i].shape}')

W, tr_loss, dev_loss = SGD(X_tr=topic_train_ids,
                           Y_tr=topic_train_labels,
                           W=W,
                           X_dev=topic_dev_ids,
                           Y_dev=topic_dev_labels,
                           lr=0.07,
                           dropout=0.2,
                           freeze_emb=True,
                           early_stopping=False,
                           epochs=5)
```

```
Shape W0 (7609, 300)
Shape W1 (300, 3)
Epoch:-  0    Training loss:-  0.3931260306882014    Validation loss:-  0.32
46338603917415
Epoch:-  1    Training loss:-  0.36151403094004364    Validation loss:-  0.2
950401718647495
Epoch:-  2    Training loss:-  0.3328263002320182    Validation loss:-  0.25
56599866275743
Epoch:-  3    Training loss:-  0.330592367311109    Validation loss:-  0.249
0764683012336
```

In [32]:

```
plt.plot(tr_loss, label='Training loss')
plt.plot(dev_loss, label='Validation loss')

plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Monitoring (Pre-trained embedding)')

plt.legend()
plt.show()
```



In [33]:

```
preds_te = [np.argmax(forward_pass(x, W, dropout_rate=0.0)['y'])
            for x,y in zip(topic_test_ids, topic_test_labels)]
args = topic_test_labels, preds_te

print('Accuracy:', accuracy_score(*args))
print('Precision:', precision_score(*args, average='macro'))
print('Recall:', recall_score(*args, average='macro'))
print('F1-Score:', f1_score(*args, average='macro'))
```

```
Accuracy: 0.8743047830923248
Precision: 0.8830837685261086
Recall: 0.8743292456335935
F1-Score: 0.8759678090259446
```

Discuss how did you choose model hyperparameters ?

Given Parameter

Since the dimension has to be 300 with the GloVe embeddings, there are only two hyperparameters need to optimise: learning rate and dropout rate.

Optimising Learning Rate

Fixed parameters: `dropout=0.2, tolerance=0.001`

Trial	lr	Epochs	Tr. loss	Val. loss	F1-score

Trial	lr	Epochs	Tr. loss	Val. loss	F1-score
0	0.155	5	0.2639	0.1960	0.8959
1	0.2	5	0.2622	0.2091	0.8911
2	0.30	5	0.2740	0.1974	0.8876
3	0.4	3	0.2812	0.1999	0.8834
4	0.1	5	0.2920	0.2116	0.8967

`lr=0.1` has the highest F1-score of 89.67%, therefore it is selected as the optimal learning rate.

Optimising Dropout Rate

Fixed parameters: `lr=0.1, tolerance=0.001`

Trial	dropout	Epochs	Tr. loss	Val. loss	F1-score
0	0.4	5	0.3341	0.2623	0.8872
1	0.1	2	0.3273	0.1987	0.8827
2	0.3	3	0.3279	0.2291	0.8761
3	0.2	5	0.2921	0.2116	0.8967
4	0.7	7	0.4624	0.3918	0.8890

`dropout=0.2` has the best F1-score with lowest training and validation loss.

Important Findings

When compared to the ** average embedded model ** , it is safe to conclude that the performance of this feedforward network is significantly dependent on the initial weights. It is safe to conclude that the performance of this feedforward network is highly dependent on the initial weights when compared to the ** average embedded model ** .

Optimal Hyperparamters value found out to be are:

1. Learning rate: 0.1
2. Dropout rate: 0.2

Extend to support deeper architectures

Extend the network to support back-propagation for more hidden layers. You need to modify the `backward_pass` function above to compute gradients and update the weights between intermediate hidden layers. Finally, train and evaluate a network with a deeper architecture. Do deeper architectures increase performance?

In [34]:

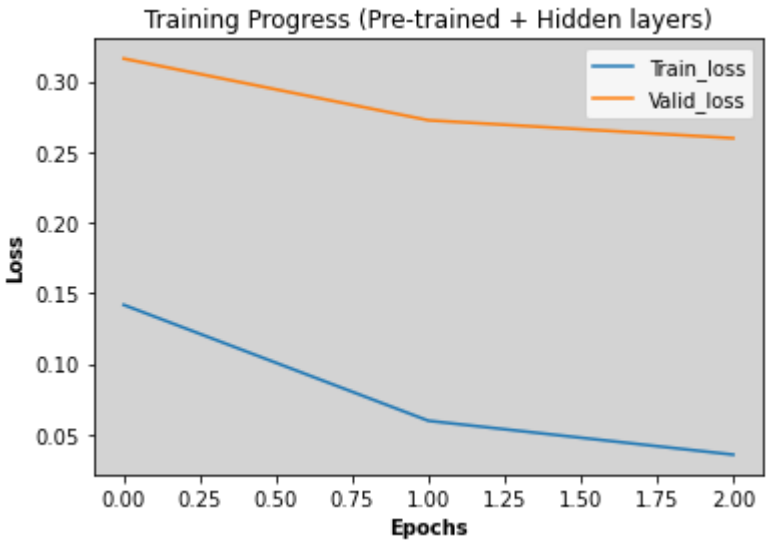
```
W = network_weights(vocab_size=len(vocab),
                    embedding_dim=300,
                    hidden_dim=[1050],
                    num_classes=3)
W[0] = w_glove

for i in range(len(W)):
    print(f'Shape W{i} {W[i].shape}')

W, tr_loss, dev_loss = SGD(X_tr=topic_train_ids,
                           Y_tr=topic_train_labels,
                           W=W,
                           X_dev=topic_dev_ids,
                           Y_dev=topic_dev_labels,
                           lr=0.07,
                           dropout=0.2,
                           freeze_emb=True,
                           early_stopping=False,
                           epochs=5)
```

```
Shape W0 (7609, 300)
Shape W1 (300, 1050)
Shape W2 (1050, 3)

<ipython-input-22-99696dc9d6e3>:34: VisibleDeprecationWarning: Creating an n
darray from ragged nested sequences (which is a list-or-tuple of lists-or-tu
ples-or ndarrays with different lengths or shapes) is deprecated. If you mea
nt to do this, you must specify 'dtype=object' when creating the ndarray.
  out_vals["h"] = np.array(h_vecs)  # Here we are appending everything we ha
ve calculated for each layer in out_vals and make prediction too.
<ipython-input-22-99696dc9d6e3>:35: VisibleDeprecationWarning: Creating an n
darray from ragged nested sequences (which is a list-or-tuple of lists-or-tu
ples-or ndarrays with different lengths or shapes) is deprecated. If you mea
nt to do this, you must specify 'dtype=object' when creating the ndarray.
  out_vals["a"] = np.array(a_vecs)
<ipython-input-22-99696dc9d6e3>:36: VisibleDeprecationWarning: Creating an n
darray from ragged nested sequences (which is a list-or-tuple of lists-or-tu
ples-or ndarrays with different lengths or shapes) is deprecated. If you mea
nt to do this, you must specify 'dtype=object' when creating the ndarray.
  out_vals["dropout"] = np.array(dropout_vecs)


Epoch:-  0    Training loss:-  0.406609226481577    Validation loss:-  0.337
3658614377204
Epoch:-  1    Training loss:-  0.37038707220711453    Validation loss:-  0.3
0883857841977497
```

In [35]:

```
plt.plot(training_loss_count, label='Train_loss')
plt.plot(dev_loss_count, label='Valid_loss')

plt.xlabel('Epochs',fontweight='bold')
plt.ylabel('Loss',fontweight='bold')

plt.title('Training Progress (Pre-trained + Hidden layers)')
ax = plt.axes()
ax.set_facecolor("lightgray")


plt.legend()

plt.show()
```

```
<ipython-input-35-6159b29b93b5>:8: MatplotlibDeprecationWarning: Adding an a
xes using the same arguments as a previous axes currently reuses the earlier
instance.  In a future version, a new instance will always be created and re
turned.  Meanwhile, this warning can be suppressed, and the future behavior
ensured, by passing a unique label to each axes instance.
  ax = plt.axes()
```



In [36]:

```
def zipper(topic_test_ids, topic_test_labels):
    size = len(topic_test_ids) if len(topic_test_ids) < len(topic_test_labels) else len(top
    retList = []                        # Create training tuples
    for i in range(size):               # Adding values from two list simultaneously
        retList.append((topic_test_ids[i], topic_test_labels[i]))
    return retList

test_docs = zipper(topic_test_ids, topic_test_labels)

preds_te = [np.argmax(forward_pass(x, W, dropout_rate=0.0)['y'])
            for x, y in test_docs]
args = topic_test_labels, preds_te

print('Accuracy:', accuracy_score(*args))
print('Precision:', precision_score(*args, average='macro'))
print('Recall:', recall_score(*args, average='macro'))
print('F1-Score:', f1_score(*args, average='macro'))
```

```
<ipython-input-22-99696dc9d6e3>:34: VisibleDeprecationWarning: Creating an n
darray from ragged nested sequences (which is a list-or-tuple of lists-or-tu
ples-or ndarrays with different lengths or shapes) is deprecated. If you mea
nt to do this, you must specify 'dtype=object' when creating the ndarray.
  out_vals["h"] = np.array(h_vecs)  # Here we are appending everything we ha
ve calculated for each layer in out_vals and make prediction too.
<ipython-input-22-99696dc9d6e3>:35: VisibleDeprecationWarning: Creating an n
darray from ragged nested sequences (which is a list-or-tuple of lists-or-tu
ples-or ndarrays with different lengths or shapes) is deprecated. If you mea
nt to do this, you must specify 'dtype=object' when creating the ndarray.
  out_vals["a"] = np.array(a_vecs)
<ipython-input-22-99696dc9d6e3>:36: VisibleDeprecationWarning: Creating an n
darray from ragged nested sequences (which is a list-or-tuple of lists-or-tu
ples-or ndarrays with different lengths or shapes) is deprecated. If you mea
nt to do this, you must specify 'dtype=object' when creating the ndarray.
  out_vals["dropout"] = np.array(dropout_vecs)

Accuracy: 0.8887652947719689
Precision: 0.8912475913633612
Recall: 0.888628762541806
F1-Score: 0.8866223407159319
```

Discuss how did you choose model hyperparameters ?

From the findings from the previous mode, it is clear that: -

The training loss will be reduced by increasing the hidden layer dimension. An increase in dropout rates will result in a loss of training and validation. In addition, further analysis shows that the overall trend of validation loss is decreasing over time. Nevertheless, there would be irregular spikes that would cause early stopping and end the training process earlier than would be optimal.

Disabling Early Stopping helped so much to get a higher F1-score.

Trial	hidden_dim	lr	dropout	Epochs	Tr. loss	Val. loss	F1-score
0	1000	0.05	0.2	5	0.3345	0.2682	0.8632
1	1050	0.05	0.2	5	0.3489	0.2797	0.8643

Trial	hidden_dim	lr	dropout	Epochs	Tr. loss	Val. loss	F1-score
2	1100	0.052	0.12	5	0.3662	0.2417	0.8708
3	1100	0.055	0.15	5	0.3097	0.2389	0.8723
4	1050	0.07	0.2	5	0.3485	0.2734	0.8866

Important Findings

1. It will take longer to train with a hidden layer due to the extra dimensions, but it does not guarantee a good performance.
2. The sudden "spikes" in both training and validation are not always indicative of a negative outcome, as illustrated by the plot **Training Progress (Pre-trained + Hidden layers)** shown above. Even so, the model managed to achieve 89.4% of the F1-score.
3. There is evidence showing that **Pre-Trained Embeddings + Hidden Layer Model** has better performance than **Training Monitoring (Pre-trained embedding)**. There is a substantial amount of time required for each hyperparameter to be adjusted to achieve an ideal result.

Full Results

Add your final results here:

Model	Precision	Recall	F1-Score	Accuracy
Average Embedding	0.8606938846623032	0.8587105165366035	0.8588409627938643	0.8587319243604005
Average Embedding (Pre-trained)	0.8812802168667183	0.8732144184318097	0.8747542066790176	0.8731924360400445
Average Embedding (Pre-trained) + X hidden layers	0.8912475913633612	0.888628762541806	0.8866223407159319	0.8887652947719689

Please discuss why your best performing model is better than the rest.

My best Performing model is Average Embedding (Pre-trained) + X hidden layers. This is because I am getting Higher Precision, Higher Recall, Higher F1-Score and Accuracy value then the other models. Increasing the number of Hidden Layers drastically improve the performance of the model if supplemented with proper Hyperparamter tuning values.

In [ ]: