[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**). **done**

Must Do

- Re-train your network by using pre-trained embeddings ([GloVe (https://nlp.stanford.edu/projects/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)](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)](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)](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)](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, re
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 [3]:
```
#read train test and dev data
train_raw = pd.read_csv('./data_topic/train.csv',names=['labels
test_raw = pd.read_csv('./data_topic/test.csv',names=['labels',
dev_raw = pd.read_csv('./data_topic/dev.csv',names=['labels','t

```

In [4]:
```
train_raw.head(5)
```

Out[4]:

	labels	text
0	1	Reuters - Venezuelans turned out early\and in ...
1	1	Reuters - South Korean police used water canno...
2	1	Reuters - Thousands of Palestinian\prisoners i...
3	1	AFP - Sporadic gunfire and shelling took place...
4	1	AP - Dozens of Rwandan soldiers flew into Suda...

In [5]:
```
test_raw.head(5)
```

Out[5]:

	labels	text
0	1	Canadian Press - VANCOUVER (CP) - The sister o...
1	1	AP - The man who claims Gov. James E. McGreeve...
2	1	NAJAF, Iraq - Explosions and gunfire rattled t...
3	1	LOURDES, France - A frail Pope John Paul II, b...
4	1	Supporters and rivals warn of possible fraud; ...

In [6]:
```
dev_raw.head(5)
```

Out[6]:

	labels	text
0	1	BAGHDAD, Iraq - An Islamic militant group that...
1	1	Parts of Los Angeles international airport are...
2	1	AFP - Facing a issue that once tripped up his ...
3	1	The leader of militant Lebanese group Hezbolla...
4	1	JAKARTA : ASEAN finance ministers ended a meet...

In [7]:
```
#convert to lowercase
train_raw['text']=train_raw['text'].str.lower()
test_raw['text']=test_raw['text'].str.lower()
dev_raw['text']=dev_raw['text'].str.lower()

#convert to list
train_raw_text = train_raw['text'].tolist()
train_raw_label = train_raw['labels'].tolist()

test_raw_text = test_raw['text'].tolist()
test_raw_label = test_raw['labels'].tolist()

dev_raw_text = dev_raw['text'].tolist()
dev_raw_label = dev_raw['labels'].tolist()
```

In [8]:
```
test_raw_text
```

```
r dueling teenager michael phelps in the 200 freestyle, won by tho
rpe...',
 " kabul (reuters) - the united states has brokered a  cease-fire
between a renegade afghan militia leader and the  embattled govern
or of the western province of herat,  washington's envoy to kabul
said tuesday.",
 'aghdad, iraq, aug. 17  a delegation of iraqis was delayed for se
curity reasons today but still intended to visit najaf to try to c
onvince a rebellious shiite cleric and his militia to evacuate a s
hrine in the holy city and end ...',
 'just what alexander downer was thinking when he declared on radi
o last friday that  quot;they could fire a missile from north kore
a to sydney quot; is unclear. the provocative remark, just days be
fore his arrival yesterday on his second visit to the north korean
...',
 'new york - stocks rose for a second straight session tuesday as
a drop in consumer prices allowed investors to put aside worries a
bout inflation, at least for the short term.    with gasoline pric
es falling to eight-month lows, the consumer price index registere
d a small drop in july, giving consumers a respite from soaring en
```

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 [9]:
```
stop_words = ['a','in','on','at','and','or',
              'to', 'the', 'of', 'an', 'by',
              'as', 'is', 'was', 'were', 'been', 'be',
              'are','for', 'this', 'that', 'these', 'those', 'y
             'it', 'he', 'she', 'we', 'they', 'will', 'have', '
              'do', 'did', 'can', 'could', 'who', 'which', 'wha
              'but', 'not', 'there', 'no', 'does', 'not', 'so',
             'his', 'her', 'they', 'them', 'from', 'with', 'its

```

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 [10]:
```
def extract_ngrams(x_raw, ngram_range=(1,3), token_pattern=r'\b
                   stop_words=[], vocab=set()):
     #convert to lowercase
    x_raw = x_raw.lower()
    tokens=[]
    ngrams=[]
    tokens_new=[]
    tuple_list=[]
    tokenexp = re.compile(token_pattern)
    #get all the unigrams based on the token pattern
    tokens = [token for token in tokenexp.findall(x_raw) if tok

    ngrams.extend(tokens)
    #print(ngrams)

    for i in range(ngram_range[0],ngram_range[1]+1):
        if i==1:
            continue
        tokens_new.extend(tokens[z:z+i] for z in range(len(toke

        for i in tokens_new:
            tuple_list.append(tuple(i))
        ngrams.extend(tuple_list)

    #print(ngrams)
    return ngrams

```

In [11]:
```
z=extract_ngrams(train_raw_text[1],
               ngram_range=(1,1),
               stop_words=stop_words,
                vocab=set())
```

In [12]:
```
train_raw_text[10]
```

Out[12]:
```
'afp - although polls show the us presidential race a virtual dead
heat, democrat john kerry appears to be gaining an edge over georg
e w. bush among the key states that could decide the outcome.'
```

In [13]:
```
z
```

Out[13]:
```
['reuters',
 'south',
 'korean',
 'police',
 'used',
 'water',
 'cannon',
 'central',
 'seoul',
 'sunday',
 'disperse',
 'least',
 'protesters',
 'urging',
 'government',
 'reverse',
 'controversial',
 'decision',
 'send',
 'more',
 'troops',
 'iraq']
```

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 [14]:

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

    df = Counter()
    ngram_counter = Counter()
    #iterate over the whole list of documents
    for doc in X_raw:
        vocab_list = extract_ngrams(doc,ngram_range=ngram_range
        #vocab=vocab_list

        #print(vocab_list)
        #calculate document frequency of ngram using counter
        df.update(set(vocab_list))

        #calculate count of each ngram in vocab
        ngram_counter.update(ngram for ngram in vocab_list if d

        #Get topN ngrams
        #create a set of vocabulary of all top N ngrams
    if keep_topN>0:
        vocab_temp=ngram_counter.most_common(keep_topN)
        vocab = {i[0] for i in vocab_temp}
    else:
        vocab = set([w for w in df if df[w]>=min_df])


    #return vocabulary, document frequency, ngram count
    return vocab, df, ngram_counter

```

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

In [15]:

```
train_vocab,train_df,train_ngram_counts = get_vocab(train_raw_t
_,dev_df, _ = get_vocab(dev_raw_text, keep_topN=5000)
_,test_df, _ = get_vocab(test_raw_text, keep_topN=5000)
```

In [16]:

```
len(train_vocab)
```

Out[16]: 8931

In [17]:
```
#function returns 2 dictionaries with vocab_id -> word and word
def vocab2word(vocab):
    v2w={}
    w2v={}
    for idx,word in enumerate(vocab):
        v2w[idx]=word
        w2v[word]=idx
    #print(v2w)

    return v2w,w2v
```

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

In [18]:
```
vocab_word = dict()
word_vocab = dict()
vocab_char={}
char_vocab={}

vocab_word,word_vocab =vocab2word(train_vocab)


```

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 [19]:
```
#get all unigrams of all the documents in the train,test,dev
train_uni_list = [extract_ngrams(x,ngram_range=(1,1), token_pat
test_uni_list = [extract_ngrams(x,ngram_range=(1,1), token_patt
dev_uni_list = [extract_ngrams(x,ngram_range=(1,1), token_patte
```

In [21]:
```
#unigrams of train document 1
train_uni_list[0]
```

Out[21]:
```
['reuters',
 'venezuelans',
 'turned',
 'out',
 'early',
 'large',
 'numbers',
 'sunday',
 'vote',
 'historic',
 'referendum',
 'either',
 'remove',
 'left',
 'wing',
 'president',
 'hugo',
 'chavez',
 'office',
 'give',
 'him',
 'new',
 'mandate',
 'govern',
 'next',
 'two',
 'years']
```

Then convert them into lists of indices in the vocabulary:

In [22]:
```
train_x=[]
dev_x=[]
test_x=[]
for x in train_uni_list:
    train_x.append([word_vocab[token] for token in x])


for i in range(len(dev_uni_list)):
    temp = []
    for x in dev_uni_list[i]:
        if x in train_vocab:
            temp.append(word_vocab[x])
    dev_x.append(temp)


for i in range(len(test_uni_list)):
    temp = []
    for x in test_uni_list[i]:
        if x in train_vocab:
            temp.append(word_vocab[x])
    test_x.append(temp)

```

In [23]:
```
train_x[2]
```

Out[23]:
```
[7383,
 1541,
 8508,
 2976,
 1831,
 5333,
 1951,
 1876,
 2,
 3254,
 3711,
 63,
 220,
 2084,
 2957,
 6220,
 2428,
 5927,
 4352,
 6166]
```

In [50]:
```
np.unique(Y_train)
```

Out[50]:
```
array([0, 1, 2])
```

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

In [24]:
```
#here i have subtracted 1 from label as it
Y_train = np.array([label-1 for label in train_raw_label])
Y_test = np.array([label-1 for label in test_raw_label])
Y_dev = np.array([label-1 for label in dev_raw_label])


Y_train.shape
```

Out[24]:
```
(2400,)
```

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 [27]:

```
def network_weights(vocab_size=1000, embedding_dim=300,
                    hidden_dim=[], num_classes=3, init_val = 0.
    #set seed for reproducability
    np.random.seed(123)
    #initializing empty weight list
    W=[]

    W.append(np.random.uniform(-init_val,init_val,size=[vocab_s
    #if no hidden layers present in the architecture
    if len(hidden_dim) == 0:
        W.append(np.random.uniform(-init_val,init_val,size=[emb
    #if hidden layers are present
    else:
        #for single hidden layer
        W.append(np.random.uniform(-init_val,init_val,size=[emb
        #for more than one hidden layer
        for i in range(len(hidden_dim)-1):
            W.append(np.random.uniform(-init_val,init_val,size=
        #last hidden layer to output layer
        W.append(np.random.uniform(-init_val,init_val,size=[hid



    return W
```

In [28]:

```
W = network_weights(vocab_size=1000,embedding_dim=300,hidden_di
```

In [29]:

```
print('W[0] (vocab X embeddings) ->', W[0].shape)
print('W[1] (embeddings X hidden_layer) ->', W[1].shape)
print('W[2] (hidden_layer X output_layer) ->', W[2].shape)
```

```
W[0] (vocab X embeddings) -> (1000, 300)
W[1] (embeddings X hidden_layer) -> (300, 3)
W[2] (hidden_layer X output_layer) -> (3, 2)
```

In [30]:

```
len(W)
```

Out[30]: 3

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 [31]:
```
def softmax(z):


    e_x = np.exp(z)
    return e_x / e_x.sum()

    #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 [32]:
```
def categorical_loss(y, y_preds):
    #print(y.shape)
    if y_preds[y]==0:
        y_preds[y]=10**-10
    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 [33]:

```
def relu(z):
    a=np.maximum(z,0)

    return a

def relu_derivative(z):
    dz = z.copy()
    dz[dz>0]=1
    dz[dz<=0]=0
    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 [34]:

```
def dropout_mask(size, dropout_rate):

    dropout_vec = np.random.choice([0., 1.], size=size, p=[drop

    return dropout_vec


```

In [160]:

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

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

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 [158]:
```
def forward_pass(x, W, dropout_rate=0.2):

    out_vals = {}
    Z_values = []
    activation_vectors = []
    dropout_vecs=[]
    emb_row = []

    hidden_layers=len(W)-2
    x_arr=np.matrix(x)

    for index in x:
        emb_row.append(W[0][index])

    input_weights = np.matrix(np.mean(np.array(emb_row), axis=0
    Z_values.append(input_weights)


    #first hidden layer
    Z1 = np.matmul(input_weights,W[1])
    A1 = relu(Z1)
    #add dropout regularization for the hidden layer output
    mask_vector = dropout_mask(A1.shape, dropout_rate)
    A1 = np.multiply(mask_vector,A1)
    Z_values.append(Z1)
    activation_vectors.append(A1)
    dropout_vecs.append(mask_vector)

    #output layer
    Z2 = np.matmul(Z1,W[2])
    A2 = softmax(Z2)

    Z_values.append(Z2)

    out_vals['z']=Z_values
    out_vals['a']=activation_vectors
    out_vals['drop_mask']=dropout_vecs
    out_vals['prediction']=np.array(A2)
    #out_vals['emb_row']=input_weights

    return out_vals

```

In [159]:
```
W = network_weights(vocab_size=30,embedding_dim=5,hidden_dim=[3

print('Weights shapes')
for i in range(0,len(W)):

    print(f'Weights[{i}] ->{W[i].shape}\n')

print(forward_pass([2,1], W, dropout_rate=0.5))

```

```
Weights shapes
Weights[0] ->(30, 5)

Weights[1] ->(5, 3)

Weights[2] ->(3, 3)

{'z': [matrix([[-0.11685776,  0.35490698,  0.06170099, -0.2296951
, -0.10491911]],
       dtype=float32), matrix([[-0.01265539,  0.10749014, -0.17042
722]], dtype=float32), matrix([[ 0.09468961, -0.05122656, -0.03315
496]], dtype=float32)], 'a': [matrix([[0., 0., 0.]])], 'drop_mask'
: [array([[1., 0., 0.]])], 'prediction': array([[0.3644022 , 0.314
9274 , 0.32067037]], dtype=float32)}
```

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 [39]:

```
def backward_pass(x, y, W, out_vals, lr=0.001, freeze_emb=False
    m = np.array(x).shape[0]
    errors = []

    #One hot encoding of label
    Y=np.zeros(3)
    Y[y]=1


    #We calculate the error of output
    Error_output =out_vals['prediction']
    Error_output =np.matrix(Error_output-Y)
    errors.append(Error_output)

    Error_weight_grad = (1.0/m)*np.matmul(np.matrix(out_vals['a

    W[-1]= W[-1] - Error_weight_grad * lr


    #hidden layer weight update
    temp_Error_h2 = np.matmul(Error_output,W[-1].T)


    Error_h2 = np.multiply(temp_Error_h2 ,relu_derivative(np.ma
    drop=dropout_mask(out_vals['z'][0].shape,0.2)
    temp_emb = np.multiply(out_vals['z'][0],drop)
    Error_W_h2 = (1.0/m) * np.dot(np.matrix(out_vals['z'][0]).T

    W[1] = W[1] - Error_W_h2 * lr


    #input layer to hidden layer weight updates
    next_gradient = np.matmul(Error_h2 , W[1].T)

    temp_z = np.multiply(next_gradient,relu_derivative(out_vals
    xt=np.ones(np.matrix(x).shape)

    next_weight_gradient = np.dot(xt.T,temp_z)


    if not freeze_emb:
        for id,i in enumerate(x):

            W[0][i] = W[0][i] - lr * next_weight_gradient[id]



    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 [189]:
```
def SGD(X_tr, Y_tr, W, X_dev=[], Y_dev=[], lr=0.001,
        dropout=0.2, epochs=5, tolerance=0.001, freeze_emb=Fals
    train_loss_history = []
    val_loss_history = []
    #set seed for reproducability
    np.random.seed(123)


    for i in range(epochs):
        #set train and val loss to 0 at beginning of each epoch
        training_loss = 0
        validation_loss = 0
        #shuffle the index of data every epoch
        index = np.arange(len(X_tr))
        np.random.shuffle(index)
        #length of training and dev data
        X_tr_LEN=len(X_tr)
        X_dev_LEN=len(X_dev)

        #feedforward pass
        for sample in range(0,X_tr_LEN):
            #get X and Y
```

```
            Y = Y_tr[index[sample]]
            X = X_tr[index[sample]]
            #forward pass
            outs = forward_pass(X, W, dropout)
            #backward pass
            W = backward_pass(X, Y, W, outs, lr=lr,freeze_emb=f


        #calculate the training loss for train data
        for sample in range(0,X_tr_LEN):
            #get X and Y
            Y = Y_tr[index[sample]]
            X = X_tr[index[sample]]
            #forward pass
            outs = forward_pass(X, W, dropout)
            #update training loss
            training_loss += categorical_loss(Y, outs['predicti
        #append the training loss to train_loss_history
        train_loss_history.append(training_loss/X_tr_LEN)


        #calculate the validation loss for dev data
        for sample in range(0,X_dev_LEN):
            #get X and Y
            Y = Y_dev[sample]
            X = X_dev[sample]
            #forward pass
            outs = forward_pass(X, W, dropout)
            #update the validation loss
            validation_loss += categorical_loss(Y, outs['predic
        #append the validation_loss to val_loss_history
        val_loss_history.append(validation_loss/X_dev_LEN)

        #print training progress if true
        if(print_progress):
        # Printing training loss and validation loss after each
            print(f"Epoch:{str(i):5} Training_loss:{str(trainin

        #Stop training if val_loss difference is less than tole
        if i>=3 and val_loss_history[i-1]-val_loss_history[i] <
            break

    #return updated Weights,training_loss_history and validatio
    return W, train_loss_history, val_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 [221]:

```
#hyperparameters combinatiom
lr=[0.01,0.001,0.005]
embedding_dim = [50,100,300]

```

In [197]:

```
param_history=[]
#SGD

# running SGD for all combinations of chosen hyperparameter val
for rate in lr:
    for embed in embedding_dim:
            # print combination of parameters
        print(f"Learning Rate:{str(rate):10} Embedding dimensio
        #print Weights dimensions
        W = network_weights(vocab_size=len(train_vocab),embeddi
                    hidden_dim=[50], num_classes=3)
            #display Weights shape
        for i in range(len(W)):
            print('Shape W'+str(i), W[i].shape)
        W, loss_tr, dev_loss = SGD(train_x,Y_train,
                                        W,
                                        X_dev=dev_x,
                                        Y_dev=Y_dev,
                                        lr=rate,
                                        dropout=0.2,
                                        freeze_emb=False,
                                        tolerance=0.0001,
                                        epochs=100,
                                        print_progress=False) #
        param_history.append([rate,embed,loss_tr,dev_loss,W])



```

```
Learning Rate:0.01       Embedding dimension:50        Hidden lay
er nodes:50
Shape W0 (8931, 50)
Shape W1 (50, 50)
Shape W2 (50, 3)
Learning Rate:0.01       Embedding dimension:100       Hidden lay
er nodes:50
Shape W0 (8931, 100)
Shape W1 (100, 50)
Shape W2 (50, 3)
Learning Rate:0.01       Embedding dimension:300       Hidden lay
er nodes:50
Shape W0 (8931, 300)
Shape W1 (300, 50)
Shape W2 (50, 3)
Learning Rate:0.001      Embedding dimension:50        Hidden lay
er nodes:50
Shape W0 (8931, 50)
```

```
Shape W1 (50, 50)
Shape W2 (50, 3)
Learning Rate:0.001      Embedding dimension:100        Hidden lay
er nodes:50
Shape W0 (8931, 100)
Shape W1 (100, 50)
Shape W2 (50, 3)
Learning Rate:0.001      Embedding dimension:300        Hidden lay
er nodes:50
Shape W0 (8931, 300)
Shape W1 (300, 50)
Shape W2 (50, 3)
Learning Rate:0.005      Embedding dimension:50         Hidden lay
er nodes:50
Shape W0 (8931, 50)
Shape W1 (50, 50)
Shape W2 (50, 3)
Learning Rate:0.005      Embedding dimension:100        Hidden lay
er nodes:50
Shape W0 (8931, 100)
Shape W1 (100, 50)
Shape W2 (50, 3)
Learning Rate:0.005      Embedding dimension:300        Hidden lay
er nodes:50
Shape W0 (8931, 300)
Shape W1 (300, 50)
Shape W2 (50, 3)
```

Plot the learning process:

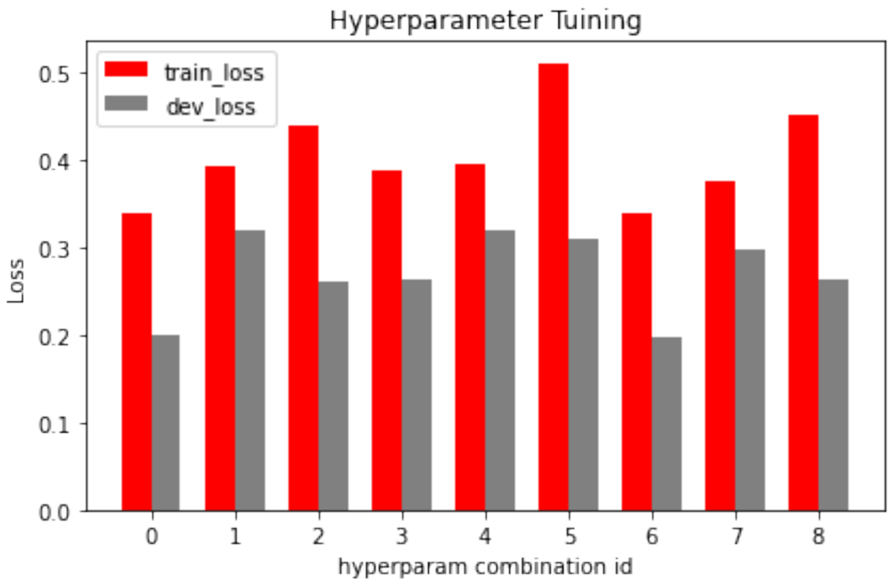
In [217]:
```
rates=[]
embed_dim=[]
loss_tr=[]
dev_loss=[]
Wghts=[]
for param in param_history:
    rates.append(param[0])
    embed_dim.append(param[1])
    loss_tr.append(param[2])
    dev_loss.append(param[3])
    Wghts.append(param[4])

df = pd.DataFrame(list(zip(rates,embed_dim)), index =np.arange(
                                              columns =['Learni

train_loss_graph=[]
dev_loss_graph=[]
for i in loss_tr:
    train_loss_graph.append(i[-1])
for i in dev_loss:
    dev_loss_graph.append(i[-1])

df
```

Out[217]:

	Learning Rate	Embedding Dim
0	0.010	50
1	0.010	100
2	0.010	300
3	0.001	50
4	0.001	100
5	0.001	300
6	0.005	50
7	0.005	100
8	0.005	300

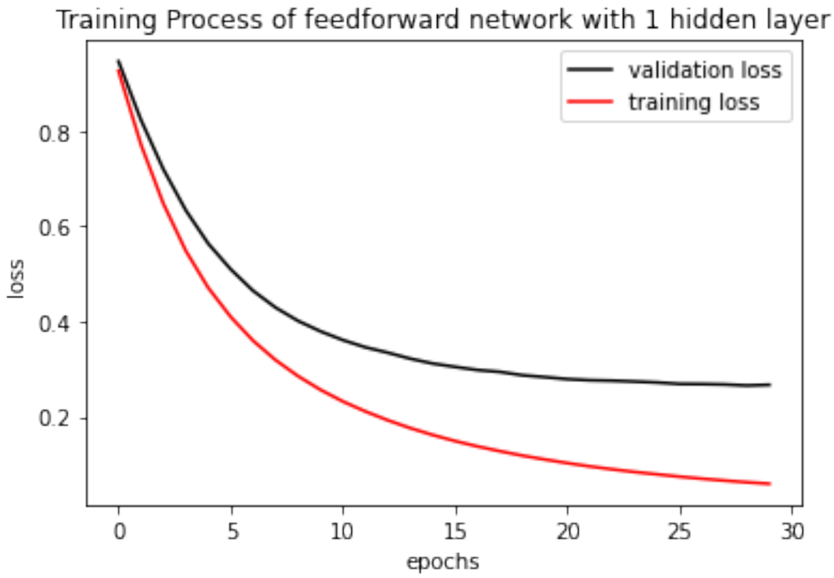
In [229]:

```
x = np.arange(len(rates))  # the label locations
width = 0.35  # the width of the bars

fig, ax = plt.subplots()
rects1 = ax.bar(x - width/2, train_loss_graph, width, label='tr
rects2 = ax.bar(x + width/2, dev_loss_graph, width, label='dev_

ax.set_ylabel('Loss')
ax.set_xlabel('hyperparam combination id')
ax.set_title('Hyperparameter Tuining')
ax.set_xticks(x)
ax.legend()


fig.tight_layout()


plt.show()
```



In [219]:

```
#training process graph of best performing model
plt.title('Training Process of feedforward network with 1 hidde
sub_axix = np.arange(len(dev_loss[0]))
plt.plot(sub_axix, dev_loss[0], color='black', label='validatio
plt.plot(sub_axix, loss_tr[0], color='red', label='training los
plt.legend()
plt.xlabel('epochs')
plt.ylabel('loss')
plt.show()
```



Compute accuracy, precision, recall and F1-Score:

In [220]:

```
final_W=Wghts[0]

preds_te = [np.argmax(forward_pass(x, final_W, dropout_rate=0.0
            for x,y in zip(test_x,Y_test)]

print('Model Precision :', precision_score(Y_test,preds_te,aver
print('Model Recall :', recall_score(Y_test,preds_te,average='m
print('Model F1-Score :', f1_score(Y_test,preds_te,average='mac
print('Model Accuracy :', accuracy_score(Y_test,preds_te))

```

```
Model Precision : 0.8630522013439728
Model Recall : 0.861111111111111
Model F1-Score : 0.8610202834545159
Model Accuracy : 0.8611111111111112
```

Discuss how did you choose model hyperparameters ?

```
chose lr=[0.01,0.001,0.005] ,embedding_dim = [50,100,300]  as my hyperparameters for fine tuning my model. Tried all the possible combinations and choose the model with lowest train and validation loss. From the graph and table above we can see that the best hyperparameters are LR=0.01 and embedding dimensions=50.
I also did tuning of hidden layer nuerons, first tried with very low value of 5 and then gradually increased the number of nuerons and chose 50 as it gives best result.
```

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 [53]:
```python
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 train_vocab:
                    emb = np.array(line.strip('\n').split()[1:]
                    w_emb[word2id[word]] +=emb
    return w_emb
```

In [55]:
```python
#Read the glove embeddings into a file using the above function
w_glove_emb = get_glove_embeddings("glove.840B.300d.zip","glove
```

In [66]:
```python
w_glove_emb.shape
```

Out[66]: (8931, 300)

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 [325]:

```
#here, embedding dim cannot be changed, so have chosen hidden_d
lr=[0.01,0.001,0.005]
hidden_dim = [100,400,700]
```

In [326]:

```
param_history=[]
#SGD

# running SGD for all combinations of chosen hyperparameter val
for rate in lr:
    for hidden in hidden_dim:
            # print combination of parameters
        print(f"Learning Rate:{str(rate):10} Embedding dimensio
        #print Weights dimensions
        W = network_weights(vocab_size=len(train_vocab),embeddi
                hidden_dim=[hidden], num_classes=3)

        #replace initialized embedding with glove embeddings
        W[0] = w_glove_emb

            #display Weights shape
        for i in range(len(W)):
            print('Shape W'+str(i), W[i].shape)
        W, loss_tr, dev_loss = SGD(train_x,Y_train,
                                    W,
                                    X_dev=dev_x,
                                    Y_dev=Y_dev,
                                    lr=rate,
                                    dropout=0.2,
                                    freeze_emb=True,
                                    tolerance=0.0001,
                                    epochs=100,
                                    print_progress=False) #
        param_history.append([rate,hidden,loss_tr,dev_loss,W])
```

```
Learning Rate:0.01       Embedding dimension:300       Hidden lay
er nodes:100
Shape W0 (8931, 300)
Shape W1 (300, 100)
Shape W2 (100, 3)
Learning Rate:0.01       Embedding dimension:300       Hidden lay
er nodes:400
Shape W0 (8931, 300)
Shape W1 (300, 400)
Shape W2 (400, 3)
Learning Rate:0.01       Embedding dimension:300       Hidden lay
er nodes:700
Shape W0 (8931, 300)
Shape W1 (300, 700)
Shape W2 (700, 3)
Learning Rate:0.001      Embedding dimension:300       Hidden lay
```

```
er nodes:100
Shape W0 (8931, 300)
Shape W1 (300, 100)
Shape W2 (100, 3)
Learning Rate:0.001      Embedding dimension:300        Hidden lay
er nodes:400
Shape W0 (8931, 300)
Shape W1 (300, 400)
Shape W2 (400, 3)
Learning Rate:0.001      Embedding dimension:300        Hidden lay
er nodes:700
Shape W0 (8931, 300)
Shape W1 (300, 700)
Shape W2 (700, 3)
Learning Rate:0.005      Embedding dimension:300        Hidden lay
er nodes:100
Shape W0 (8931, 300)
Shape W1 (300, 100)
Shape W2 (100, 3)
Learning Rate:0.005      Embedding dimension:300        Hidden lay
er nodes:400
Shape W0 (8931, 300)
Shape W1 (300, 400)
Shape W2 (400, 3)
Learning Rate:0.005      Embedding dimension:300        Hidden lay
er nodes:700
Shape W0 (8931, 300)
Shape W1 (300, 700)
Shape W2 (700, 3)
```

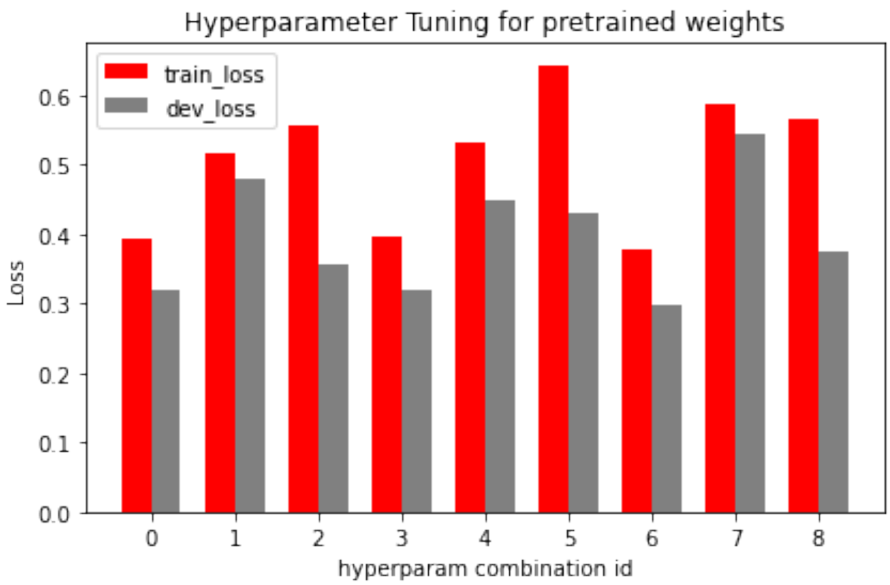
In [327]:

```
rates=[]
hidden=[]
loss_tr=[]
dev_loss=[]
Wghts=[]
for param in param_history:
    rates.append(param[0])
    hidden.append(param[1])
    loss_tr.append(param[2])
    dev_loss.append(param[3])
    Wghts.append(param[4])

df = pd.DataFrame(list(zip(rates,hidden)), index =np.arange(len
                                                             columns =['Learni

train_loss_graph=[]
dev_loss_graph=[]
for i in loss_tr:
    train_loss_graph.append(i[-1])
for i in dev_loss:
    dev_loss_graph.append(i[-1])

df
```

Out[327]:

	Learning Rate	hidden Dim
0	0.010	100
1	0.010	400
2	0.010	700
3	0.001	100
4	0.001	400
5	0.001	700
6	0.005	100
7	0.005	400
8	0.005	700

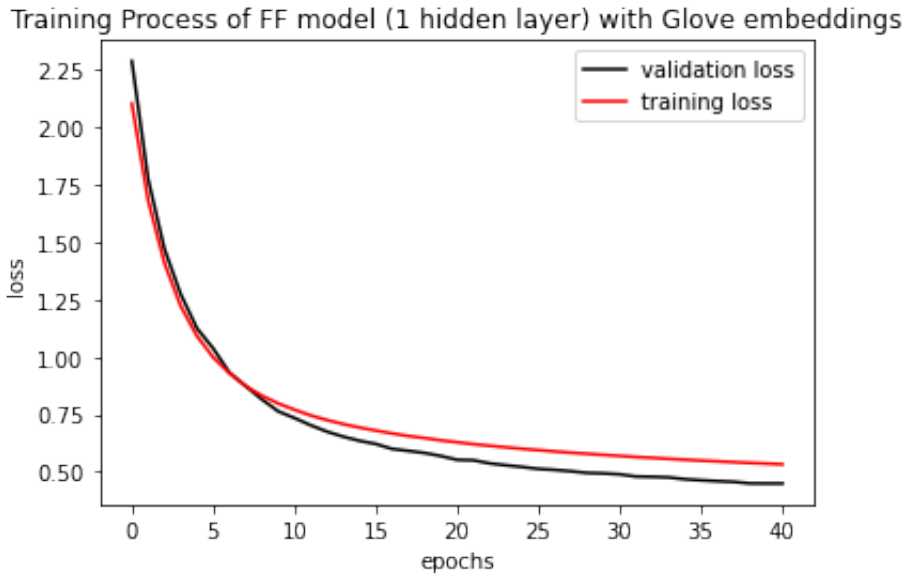
In [328]:
```
x = np.arange(len(rates))  # the label locations
width = 0.35  # the width of the bars

fig, ax = plt.subplots()
rects1 = ax.bar(x - width/2, train_loss_graph, width, label='tr
rects2 = ax.bar(x + width/2, dev_loss_graph, width, label='dev_

ax.set_ylabel('Loss')
ax.set_xlabel('hyperparam combination id')
ax.set_title('Hyperparameter Tuning for pretrained weights')
ax.set_xticks(x)
ax.legend()


fig.tight_layout()


plt.show()
```



In [329]:
```
final_W=Wghts[6]

preds_te = [np.argmax(forward_pass(x, final_W, dropout_rate=0.0
            for x,y in zip(test_x,Y_test)]

print('Model Precision :', precision_score(Y_test,preds_te,aver
print('Model Recall :', recall_score(Y_test,preds_te,average='m
print('Model F1-Score :', f1_score(Y_test,preds_te,average='mac
print('Model Accuracy :', accuracy_score(Y_test,preds_te))
```

```
Model Precision : 0.8454087585749361
Model Recall : 0.8422222222222221
Model F1-Score : 0.8428213588992411
Model Accuracy : 0.8422222222222222
```

In [331]:
```
plt.title('Training Process of FF model (1 hidden layer) with G
sub_axix = np.arange(len(dev_loss[4]))
plt.plot(sub_axix, dev_loss[4], color='black', label='validatio
plt.plot(sub_axix, loss_tr[4], color='red', label='training los
plt.legend()
plt.xlabel('epochs')
plt.ylabel('loss')
plt.show()
```



Discuss how did you choose model hyperparameters ?

chose lr=[0.01,0.001,0.005] ,hidden_dim = [5,10,20] as my hyperparameters for fine tuning my model. Tried all the possible combinations and choose the model with lowest train and validation loss. From the graph and table above we can see that the best hyperparameters are LR=0.005 and hidden dimensions=10. Here, we cannot fine tune the embedding dimension as they are pretrained.

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 [269]:

```python
def forward_pass_new(x, W, dropout_rate=0.2):

    out_vals = {}
    Z_values = []
    activation_vectors = []
    dropout_vecs=[]
    emb = []
    x_arr=np.matrix(x)


    for index in x:
        emb.append(W[0][index])

    input_weights = np.matrix(np.mean(np.array(emb), axis=0))
    Z_values.append(input_weights)


    #first hidden layer
    Z1 = np.matmul(input_weights,W[1])
    #hidden layer activation function
    A1 = relu(Z1)
    #apply dropout regularization
    mask_vector = dropout_mask(A1.shape, dropout_rate)
    A1 = np.multiply(mask_vector,A1)
    Z_values.append(Z1)
    activation_vectors.append(A1)
    dropout_vecs.append(mask_vector)

    #Second hidden layer
    Z2 = np.matmul(Z1,W[2])
    #hidden layer activation function
    A2 = relu(Z2)
    #apply dropout regularization
    mask_vector = dropout_mask(A2.shape, dropout_rate)
    A2 = np.multiply(mask_vector,A2)
    Z_values.append(Z2)
    activation_vectors.append(A2)
    dropout_vecs.append(mask_vector)

    #output layer
    Z3 = np.matmul(Z2,W[3])
    #output activation function
    A3 = softmax(Z3)
    Z_values.append(Z3)

    out_vals['z']=Z_values
    out_vals['a']=activation_vectors
    out_vals['drop_mask']=dropout_vecs
    out_vals['prediction']=np.array(A3)
    #out_vals['emb']=input_weights

    return out_vals
```

In [118]:

```
def backward_pass_new(x, y, W, out_vals, lr=0.001, freeze_emb=F
    m = np.array(x).shape[0]
    errors = []

    #One hot encoding of label
    Y=np.zeros(3)
    Y[y]=1

    #We calculate the error of output
    Error_output =out_vals['prediction']
    Error_output =np.matrix(Error_output-Y)
    errors.append(Error_output)
    #Calculate the weight gradient for weights between output l
    Error_weight_grad = (1.0/m)*np.matmul(np.matrix(out_vals['a
    #update weights
    W[-1]= W[-1] - Error_weight_grad * lr


    #2nd hidden layer weight update
    temp_Error_h2 = np.matmul(Error_output,W[-1].T)
    Error_h2 = np.multiply(temp_Error_h2 ,relu_derivative(np.ma
    drop=dropout_mask(out_vals['a'][0].shape,0.2)
    temp_emb = np.multiply(out_vals['a'][0],drop)
    #calculate the weight gradient for weights between 2nd hidd
    Error_W_h2 = (1.0/m) * np.dot(np.matrix(out_vals['a'][0]).T
    #update weights
    W[2] = W[2] - Error_W_h2 * lr


    #1st hidden layer update
    temp_Error_h1 = np.matmul(Error_h2,W[-2].T)
    Error_h1 = np.multiply(temp_Error_h1 ,relu_derivative(np.ma
    drop=dropout_mask(out_vals['z'][0].shape,0.2)
    temp_emb = np.multiply(out_vals['z'][0],drop)
    #calculate the weight gradient for weights between 1st hidd
    Error_W_h1 = (1.0/m) * np.dot(np.matrix(out_vals['z'][0]).T

    W[1] = W[1] - Error_W_h1 * lr


    #input layer to hidden layer weight updates
    next_gradient = np.matmul(Error_h1 , W[1].T)
    temp_z = np.multiply(next_gradient,relu_derivative(out_vals
    xt=np.ones(np.matrix(x).shape)
    #calculate weight gradient for embedding weights
    next_weight_gradient = np.dot(xt.T,temp_z)

    #update embedding weights if freeze_emb is false
    if freeze_emb==False:
        for id,i in enumerate(x):
            W[0][i] = W[0][i] - lr * next_weight_gradient[id]
```

```
    return W
```

In [277]:

```
_new(X_tr, Y_tr, W, X_dev=[], Y_dev=[], lr=0.001,
 dropout=0.2, epochs=5, tolerance=0.001, freeze_emb=False, print_pro
t seed for reproducability
random.seed(123)

in_loss_history = []
_loss_history = []
 i in range(epochs):

 #shuffle the index of data every epoch
 idx = np.arange(len(X_tr))
 np.random.shuffle(idx)
 train_loss = 0
 validation_loss = 0
 X_tr_LEN = len(X_tr)
 X_dev_LEN = len(X_dev)

 #feedforward pass
 for sample in range(X_tr_LEN):
     t_label = Y_tr[idx[sample]]
     t_data = X_tr[idx[sample]]
     output = forward_pass_new(t_data, W, dropout)
     W = backward_pass_new(t_data, t_label, W, output, lr=lr,freeze_


 #calculate the training loss
 for sample in range(X_tr_LEN):
     t_label = Y_tr[idx[sample]]
     t_data = X_tr[idx[sample]]
     #print('label--->')
     #print(temp_label)
     output = forward_pass_new(t_data, W, dropout)
     #print(output['prediction'])
     train_loss += categorical_loss(t_label, output['prediction'][0]
 #append the training loss to train_loss_history
 train_loss_history.append(train_loss/X_tr_LEN)


 #calculate the validation loss
 for sample in range(X_dev_LEN):
     t_label = Y_dev[sample]
     t_data = X_dev[sample]
     output = forward_pass_new(t_data, W, dropout)

     validation_loss += categorical_loss(t_label, output['prediction
 #append the validation_loss to val_loss_history
 val_loss_history.append(validation_loss/X_dev_LEN)

 if print_progress:
 # Printing loss after each epoch
     print(f"Epochs:{str(i):5} Training loss:{str(train_loss/X_tr_LE
```

```
#Stop training if val_loss difference is less than tolerance
if i>=3 and val_loss_history[i-1]-val_loss_history[i] < tolerance:
    break


urn W, train_loss_history, val_loss_history
```

In [278]:

```
#hyperparameters combinatiom
lr=[0.01,0.001,0.005]
embedding_dim = [50,100,300]
```

In [283]:

```
param_history=[]
#SGD

# running SGD for all combinations of chosen hyperparameter valu
for rate in lr:
    for embed in embedding_dim:
            # print combination of parameters
        print(f"Learning Rate:{str(rate):10} Embedding dimension
        #print Weights dimensions
        W = network_weights(vocab_size=len(train_vocab),embeddin
                hidden_dim=[20,5], num_classes=3)
            #display Weights shape
        for i in range(len(W)):
            print('Shape W'+str(i), W[i].shape)
        W, loss_tr, dev_loss = SGD_new(train_x,Y_train,
                                        W,
                                        X_dev=dev_x,
                                        Y_dev=Y_dev,
                                        lr=rate,
                                        dropout=0.2,
                                        freeze_emb=False,
                                        tolerance=0.0001,
                                        epochs=100,
                                        print_progress=False) #
        param_history.append([rate,embed,loss_tr,dev_loss,W])
```

```
Learning Rate:0.01       Embedding dimension:50        Hidden lay
er nodes:50
Shape W0 (8931, 50)
Shape W1 (50, 20)
Shape W2 (20, 5)
Shape W3 (5, 3)
Learning Rate:0.01       Embedding dimension:100       Hidden lay
er nodes:50
Shape W0 (8931, 100)
Shape W1 (100, 20)
```

```
Shape W2 (20, 5)
Shape W3 (5, 3)
Learning Rate:0.01       Embedding dimension:300        Hidden lay
er nodes:50
Shape W0 (8931, 300)
Shape W1 (300, 20)
Shape W2 (20, 5)
Shape W3 (5, 3)
Learning Rate:0.001      Embedding dimension:50         Hidden lay
er nodes:50
Shape W0 (8931, 50)
Shape W1 (50, 20)
Shape W2 (20, 5)
Shape W3 (5, 3)
Learning Rate:0.001      Embedding dimension:100        Hidden lay
er nodes:50
Shape W0 (8931, 100)
Shape W1 (100, 20)
Shape W2 (20, 5)
Shape W3 (5, 3)
Learning Rate:0.001      Embedding dimension:300        Hidden lay
er nodes:50
Shape W0 (8931, 300)
Shape W1 (300, 20)
Shape W2 (20, 5)
Shape W3 (5, 3)
Learning Rate:0.005      Embedding dimension:50         Hidden lay
er nodes:50
Shape W0 (8931, 50)
Shape W1 (50, 20)
Shape W2 (20, 5)
Shape W3 (5, 3)
Learning Rate:0.005      Embedding dimension:100        Hidden lay
er nodes:50
Shape W0 (8931, 100)
Shape W1 (100, 20)
Shape W2 (20, 5)
Shape W3 (5, 3)
Learning Rate:0.005      Embedding dimension:300        Hidden lay
er nodes:50
Shape W0 (8931, 300)
Shape W1 (300, 20)
Shape W2 (20, 5)
Shape W3 (5, 3)
```

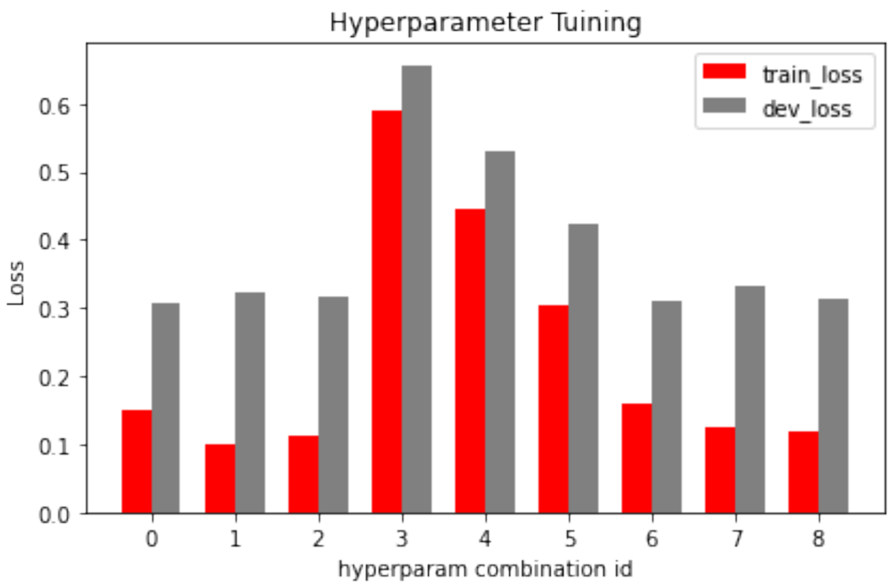
In [284]:
```
rates=[]
embed_dim=[]
loss_tr=[]
dev_loss=[]
Wghts=[]
for param in param_history:
    rates.append(param[0])
    embed_dim.append(param[1])
    loss_tr.append(param[2])
    dev_loss.append(param[3])
    Wghts.append(param[4])

df = pd.DataFrame(list(zip(rates,embed_dim)), index =np.arange(
                                                              columns =['Learni

train_loss_graph=[]
dev_loss_graph=[]
for i in loss_tr:
    train_loss_graph.append(i[-1])
for i in dev_loss:
    dev_loss_graph.append(i[-1])

df
```

Out[284]:

	Learning Rate	Embedding Dim
0	0.010	50
1	0.010	100
2	0.010	300
3	0.001	50
4	0.001	100
5	0.001	300
6	0.005	50
7	0.005	100
8	0.005	300

In [285]:

```
x = np.arange(len(rates))  # the label locations
width = 0.35  # the width of the bars

fig, ax = plt.subplots()
rects1 = ax.bar(x - width/2, train_loss_graph, width, label='tr
rects2 = ax.bar(x + width/2, dev_loss_graph, width, label='dev_

ax.set_ylabel('Loss')
ax.set_xlabel('hyperparam combination id')
ax.set_title('Hyperparameter Tuining')
ax.set_xticks(x)
ax.legend()


fig.tight_layout()


plt.show()
```



In [291]:

```
final_W=Wghts[0]

preds_te = [np.argmax(forward_pass_new(x, final_W, dropout_rate
            for x,y in zip(test_x,Y_test)]

print('Model Precision :', precision_score(Y_test,preds_te,aver
print('Model Recall :', recall_score(Y_test,preds_te,average='m
print('Model F1-Score :', f1_score(Y_test,preds_te,average='mac
print('Model Accuracy :', accuracy_score(Y_test,preds_te))
```

```
Model Precision : 0.856175527903469
Model Recall : 0.8522222222222222
Model F1-Score : 0.8516108108108109
Model Accuracy : 0.8522222222222222
```

In [ ]:

```
#training process graph of best performing model
plt.title('Training Process of FF model (2 hidden layer)')
sub_axix = np.arange(len(dev_loss[0]))
plt.plot(sub_axix, dev_loss[0], color='black', label='validatio
plt.plot(sub_axix, loss_tr[0], color='red', label='training los
plt.legend()
plt.xlabel('epochs')
plt.ylabel('loss')
plt.show()
```

In [ ]:

```

```

In [318]:

```
#here, embedding dim cannot be changed, so have chosen hidden_d
lr=[0.01,0.001,0.005]
hidden_dim = [[5,3],[5,5],[10,10]]
```

In [320]:

```
param_history=[]
#SGD

# running SGD for all combinations of chosen hyperparameter val
for rate in lr:
    for hidden in hidden_dim:

            # print combination of parameters
        print(f"Learning Rate:{str(rate):10} Embedding dimensio
        #print Weights dimensions
        W = network_weights(vocab_size=len(train_vocab),embeddi
                hidden_dim=hidden, num_classes=3)

        #replace initialized embedding with glove embeddings
        W[0] = w_glove_emb

            #display Weights shape
        for i in range(len(W)):
            print('Shape W'+str(i), W[i].shape)
        W, loss_tr, dev_loss = SGD_new(train_x,Y_train,
                                        W,
                                        X_dev=dev_x,
                                        Y_dev=Y_dev,
                                        lr=rate,
                                        dropout=0.2,
                                        freeze_emb=True,
                                        tolerance=0.0001,
                                        epochs=100,
                                        print_progress=False) #
        param_history.append([rate,hidden,loss_tr,dev_loss,W])


```

```
Learning Rate:0.001      Embedding dimension:300        Hidden lay
er nodes:[5, 5]
Shape W0 (8931, 300)
Shape W1 (300, 5)
Shape W2 (5, 5)
Shape W3 (5, 3)
Learning Rate:0.001      Embedding dimension:300        Hidden lay
er nodes:[10, 10]
Shape W0 (8931, 300)
Shape W1 (300, 10)
Shape W2 (10, 10)
Shape W3 (10, 3)
Learning Rate:0.005      Embedding dimension:300        Hidden lay
er nodes:[5, 3]
Shape W0 (8931, 300)
Shape W1 (300, 5)
Shape W2 (5, 3)
Shape W3 (3, 3)
Learning Rate:0.005      Embedding dimension:300        Hidden lay
er nodes:[5, 5]
```

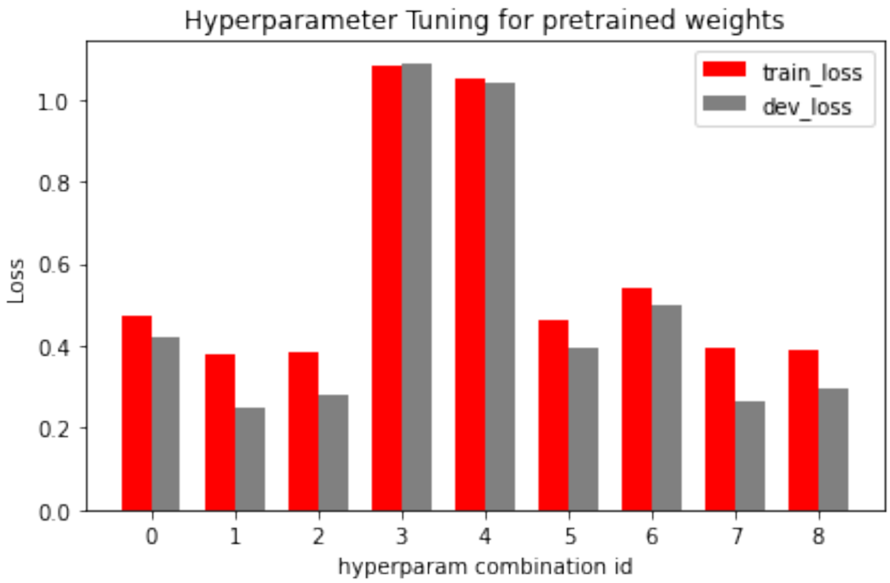
In [321]:

```
rates=[]
hidden=[]
loss_tr=[]
dev_loss=[]
Wghts=[]
for param in param_history:
    rates.append(param[0])
    hidden.append(param[1])
    loss_tr.append(param[2])
    dev_loss.append(param[3])
    Wghts.append(param[4])

df = pd.DataFrame(list(zip(rates,hidden)), index =np.arange(len
                                             columns =['Learni

train_loss_graph=[]
dev_loss_graph=[]
for i in loss_tr:
    train_loss_graph.append(i[-1])
for i in dev_loss:
    dev_loss_graph.append(i[-1])

df

W
```

Out[321]:

```
[array([[ 0.54645997, -0.37663001,  0.041767  , ...,  0.74782002,
          0.26929   , -0.20068   ],
        [ 0.026412  , -0.037006  , -0.242     , ...,  0.21589001,
          0.31134   , -0.14387999],
        [-0.24186   , -0.13191   ,  0.33013001, ..., -0.43035999,
         -0.20152   ,  0.20290001],
        ...,
        [ 0.27553999,  0.12768   ,  0.57435   , ...,  0.0069836 ,
          0.06843   , -0.69524997],
        [ 0.094195  , -0.1988    , -0.13933   , ...,  0.23405001,
         -0.3698    ,  0.10991   ],
        [-0.16414   ,  0.27024001,  0.22702   , ..., -0.47720999,
         -0.44316   ,  0.12215   ]]),
 matrix([[ 0.14251736,  0.09757374, -0.05589359, ..., -0.11605082,
           0.22973698, -0.2070698 ],
         [ 0.36576488, -0.00808336,  0.00835774, ...,  0.46019585,
           0.20034172, -0.12602231],
         [-0.04454451,  0.0741214 , -0.52144565, ..., -0.35561113,
          -0.38806303, -0.18954414],
```

In [322]:
```
x = np.arange(len(rates))  # the label locations
width = 0.35  # the width of the bars

fig, ax = plt.subplots()
rects1 = ax.bar(x - width/2, train_loss_graph, width, label='tr
rects2 = ax.bar(x + width/2, dev_loss_graph, width, label='dev_

ax.set_ylabel('Loss')
ax.set_xlabel('hyperparam combination id')
ax.set_title('Hyperparameter Tuning for pretrained weights')
ax.set_xticks(x)
ax.legend()


fig.tight_layout()


plt.show()
```

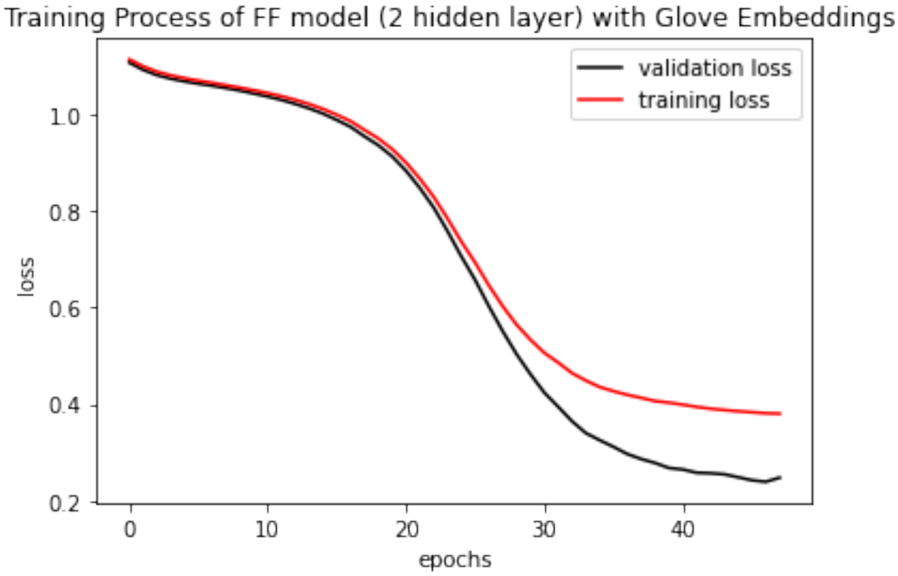


In [323]:
```
final_W=Wghts[1]

preds_te = [np.argmax(forward_pass_new(x, final_W, dropout_rate
            for x,y in zip(test_x,Y_test)]

print('Model Precision :', precision_score(Y_test,preds_te,aver
print('Model Recall :', recall_score(Y_test,preds_te,average='m
print('Model F1-Score :', f1_score(Y_test,preds_te,average='mac
print('Model Accuracy :', accuracy_score(Y_test,preds_te))
```

```
Model Precision : 0.8620155658099747
Model Recall : 0.8588888888888889
Model F1-Score : 0.8593151662047184
Model Accuracy : 0.8588888888888889
```

In [324]:

```
plt.title('Training Process of FF model (2 hidden layer) with G
sub_axix = np.arange(len(dev_loss[1]))
plt.plot(sub_axix, dev_loss[1], color='black', label='validatio
plt.plot(sub_axix, loss_tr[1], color='red', label='training los
plt.legend()
plt.xlabel('epochs')
plt.ylabel('loss')
plt.show()
```



Discuss how did you choose model hyperparameters ?

chose lr=[0.01,0.001,0.005] ,hidden_dim = [[5,3],[5,5],[10,10]] as my hyperparameters for fine tuning my model. Tried all the possible combinations and choose the model with lowest train and validation loss. From the graph and table above we can see that the best hyperparameters are LR=0.01 and hidden dimensions=[5,3].

Full Results

Add your final results here:

Model	Precision	Recall	F1-Score	Accuracy
Average Embedding	0.826	0.823	0.823	0.823
Average Embedding (Pre-trained)	0.8454	0.8422	0.8428	0.8422
Average Embedding (Pre-trained) + X hidden layers	0.864	0.862	0.862	0.862

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

My best perfroming model is avrage embedding (Pre-trained) + X hidden layers. The model has 2 hidden layers and is trained with pre-trained weights (Glove), more hidden layers in model will learn more features and hence will be best performing.

In [ ]:

```
```