The goal of this assignment is to develop a Feedforward network for text 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 (**4 marks**)
  - Perform a **Backward pass** to compute gradients and update all sets of weights (**4 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 hyperparam combination (**2 marks**).
- After training the model, plot the learning process (i.e. training and validation loss in each epoch) using a line plot and report accuracy.
- 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**).
- **BONUS:** Extend you Feedforward network by adding more hidden layers (e.g. one more). 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 (**+2 extra marks**)

Data

The data you will use for Task 2 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/2/library/index.html), NumPy, SciPy 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 etc.. You are allowed to re-use your code from Assignment 1.

Please make sure to comment your code. You should also mention if you've used Windows to write and test your code. There is no single correct answer on what your accuracy should be, but correct implementations usually achieve F1 of ~75-80% and ~85% without and with using pre-trained embeddings respectively.

This assignment will be marked out of 20. It is worth 20% of your final grade in the module. If you implement the bonus question you can get up to 2 extra points but your final grade will be capped at 20.

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]:

```
data_tr = pd.read_csv("./data_topic/train.csv",header=None, names=['label','text'])
data_te = pd.read_csv("./data_topic/dev.csv",header=None, names=['label','text'])
data_dev = pd.read_csv("./data_topic/test.csv",header=None, names=['label','text'])
```

In [3]:

```
X_tr_raw = list(data_tr['text'])
X_test_raw = list(data_te['text'])
X_dev_raw = list(data_dev['text'])
```

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 [4]:

```
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', 'you', 'i', 'if',
             'it', 'he', 'she', 'we', 'they', 'will', 'have', 'has',
              'do', 'did', 'can', 'could', 'who', 'which', 'what',
              'but', 'not', 'there', 'no', 'does', 'not', 'so', 've', 'their',
             '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 [5]:

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

    if ngram_range[1] == 1:
        return x

    # 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 lenghts
        # but zip ignores redundant elements at the end of the second list
        # Alternativel, 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 [6]:

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

    df = Counter()
    ngram_counts = Counter()
    vocab = set()

    # interate through each raw text
    for x in X_raw:

        x_ngram = extract_ngrams(x, ngram_range=ngram_range, token_pattern=r'\b[A-Za-z][A-Za-z]+\b'

        #update doc and ngram frequencies
        df.update(list(set(x_ngram)))
        ngram_counts.update(x_ngram)

    # obtain a vocabulary as a set.
    # Keep elements with doc frequency > minimum doc freq (min_df)
    # Note that df contains all te
    vocab = set([w for w in df if df[w]>=min_df])

    # keep the top N most freqent
    if keep_topN>0:
        vocab = set([w[0] for w in ngram_counts.most_common(keep_topN) if w[0] in vocab])


    return vocab, df, ngram_counts
```

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

In [7]:

```
vocab, df, ngram_counts = get_vocab(X_tr_raw, ngram_range=(1, 1), stop_words=stop_words)
```

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

In [8]:

```
# generate word and corresponding id
id2word = dict(enumerate(vocab))

word2id = {}
i = 0
for i, w in enumerate(list(vocab)):
    word2id[w] = i
```

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 [9]:

```
X_train_ngram = []
X_dev_ngram = []
X_test_ngram = []

# generate 3 file's ngrams list
for i in X_tr_raw:
    X_train_ngram.append(extract_ngrams(i, ngram_range=(1,3), token_pattern=r'\b[A-Za-z][A-Za-z]+\b
for i in X_dev_raw:
    X_dev_ngram.append(extract_ngrams(i, ngram_range=(1,3), token_pattern=r'\b[A-Za-z][A-Za-z]+\b',
for i in X_test_raw:
    X_test_ngram.append(extract_ngrams(i, ngram_range=(1,3), token_pattern=r'\b[A-Za-z][A-Za-z]+\b'
```

Then convert them into lists of indices in the vocabulary:

In [10]:

```
def uni2id(data, word2id):
    result = []
    for i in range(len(data)):
      row = []
      for j in range(len(data[i])):
          row.append(word2id[data[i][j]])
      result.append(row)
    return result


X_tr = uni2id(X_train_ngram, word2id)
X_dev = uni2id(X_dev_ngram, word2id)
X_te = uni2id(X_test_ngram,word2id)
```

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

In [11]:

```
# -1 to make more convinent when we use this in future steps
Y_tr = data_tr['label'].values -1
Y_dev = data_dev['label'].values - 1
Y_te = data_te['label'].values -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:

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

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:

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

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

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 (for the Bonus). Empty if there are no hidden layers between the average embedding and the output layer
- `num_clusses` : 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)

See the examples below for expected outputs. 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 [12]:

```
def network_weights(vocab_size=1000, embedding_dim=300,
                    hidden_dim=[], num_classes=3, init_val = 0.5):
  output_dim = embedding_dim
  W = []
  W.append(np.random.uniform(-0.1, 0.1, size=(vocab_size, embedding_dim)).astype(np.float32))  # pr
  if hidden_dim:
      W.append(np.random.uniform(-0.1, 0.1, size=(embedding_dim, hidden_dim[0])).astype(np.float32)
      output_dim = hidden_dim[0]

  W.append(np.random.uniform(-0.1, 0.1, size=(output_dim, num_classes)).astype(np.float32)) # proce
  return W
```

In [13]:

```
W = network_weights(vocab_size=5,embedding_dim=10,hidden_dim=[], num_classes=2)

print('W_emb:', W[0].shape)
print('W_out:', W[1].shape)
```

```
W_emb: (5, 10)
W_out: (10, 2)
```

In [14]:

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

In [15]:

```
print('W_emb:', W[0].shape)
print('W_h1:', W[1].shape)
print('W_out:', W[2].shape)
```

```
W_emb: (3, 4)
W_h1: (4, 2)
W_out: (2, 2)
```

In [16]:

```
W[0]
```

Out[16]:

```
array([[-0.08085749,  0.07706536,  0.02544979,  0.04468327],
       [-0.09677416,  0.01888638,  0.01135704, -0.06820807],
       [-0.06938589,  0.03910591, -0.03624671,  0.03839406]],
      dtype=float32)
```

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):
    smax = np.exp(z) / np.sum(np.exp(z))
    return smax
```

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):
    loss = -np.log(y_preds[y])
    return loss
```

In [19]:

```
# example for 5 classes

y = 2 #true label
y_preds = softmax(np.array([[-2.1,1.,0.9,-1.3,1.5]]))[0]

print('y_preds: ',y_preds)
print('loss:', categorical_loss(y, y_preds))
```

```
y_preds:  [0.01217919 0.27035308 0.24462558 0.02710529 0.44573687]
loss: 1.40802648485675
```

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):

$$\text{relu\_derivative}(z_i) = \begin{cases} 0, & \text{if } z_i <= 0. \\ 1, & \text{otherwise.} \end{cases}$$

Note that both functions take as input a vector z

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

In [20]:

```
def relu(z):

  a = np.array([max(x, 0) for x in z])
  return a

def relu_derivative(z):
    return 1 * (z > 0)
```

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 [21]:

```

def dropout_mask(size, dropout_rate):
  dropout_vec = np.ones(size)
  indexs = random.sample(range(0, size), int(size*dropout_rate))
  for i in indexs:

      dropout_vec[i] = 0

  return dropout_vec
```

In [22]:

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

```
[0. 1. 1. 1. 0. 1. 1. 1. 1. 1.]
[1. 0. 1. 1. 1. 1. 0. 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 [23]:

```
def predict(h, W):
  preds = softmax(np.matmul(h, W))
  return preds
```

In [24]:

```
def forward_pass(x, W, dropout_rate=0.2):
    out_vals = {}
    h_vecs = []
    a_vecs = []
    dropout_vecs = []
    # produce dropout_mask vectors based on size of hidden layer
    dropout_vecs.append(dropout_mask(W[0].shape[1], dropout_rate))
    # get the weights mapping with x and get mean value
    h = np.sum(W[0][x], axis=0)/len(x)
    h_vecs.append(np.array(h, dtype=np.float32))
    # put relu(h) values into a_vecs
    relu_h = relu(h_vecs[0])
    a_vecs.append(np.array(relu_h, dtype=np.float32))# h0
    # loop all hidden layer
    hid_num = len(W)-1
    for hl in range(1, hid_num):
        # dropout for regularisation.
        dropout = dropout_mask(W[hl].shape[1], dropout_rate)
        h_t = np.dot(h_vecs[hl-1], W[hl])
        # put the dropout into vectors
        h_vecs.append(dropout*h_t)
        a_vecs.append(relu(h_vecs[hl]))
        dropout_vecs.append(dropout)

    # make prediction
    if len(W)>1:
        y_pred = predict(h_vecs[hid_num-1], W[-1])
    else:
        y_pred = predict(h_vecs[0], W[-1])

    # compute cross-entropy loss
    out_vals['h'] = h_vecs
    out_vals['a'] = a_vecs
    out_vals['dropout_vec'] = dropout_vecs
    out_vals['y'] = y_pred
    return out_vals
```

In [25]:

```
W = network_weights(vocab_size=3,embedding_dim=4,hidden_dim=[5], num_classes=2)

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

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

```
Shape W0 (3, 4)
Shape W1 (4, 5)
Shape W2 (5, 2)

{'h': [array([-0.06532356, -0.0525649 , -0.01648912,  0.03281889], dtype=float32), a
rray([-0.        ,  0.00132905, -0.00631938,  0.        ,  0.00368075])], 'a': [arra
y([0.        , 0.        , 0.        , 0.03281889], dtype=float32), array([-0.
,  0.00132905,  0.        ,  0.        ,  0.00368075])], 'dropout_vec': [array([0.,
1., 0., 1.]), array([0., 1., 1., 0., 1.])], 'y': array([0.50001371, 0.49998629])}
```

The `backward_pass` function computes the gradients and update 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 [26]:

```
def backward_pass(x, y, W, out_vals, lr=0.001, freeze_emb=False):
    # Derivation of cross-entropy softmax
    y_pd = out_vals['y']
    y2 = np.zeros(y_pd.shape[-1])
    y2[y] = 1
    dy = y_pd - y2
    dy = np.reshape(dy, (1,dy.shape[-1]))
    n = len(W)
    # process hidden layer part
    for i in range(n-1,0,-1):
        # get w gradient
        w_grad = out_vals['a'][i-1].T[:,None].dot(dy)
        # get h gradient
        h_grad = dy.dot(W[i].T)[0]
        #  update w value
        W[i] = W[i] - lr*w_grad
        dy = relu_derivative(h_grad)
        # change it to (1, 300)
        dy = np.reshape(dy, (1,dy.shape[-1]))
    if not freeze_emb:
        dy = np.reshape(dy, (dy.shape[-1]))
        embed_grad = relu_derivative(dy)
        for id in x:
            W[0][id] = W[0][id] - embed_grad * lr
    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 [27]:

```
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, print_progress=True):
    # get training and development data numbers
    X_tr_nums = len(X_tr)
    X_dev_nums = len(X_dev)
    training_loss_history = []
    validation_loss_history = []
    initial_loss = np.inf
    for i in range(epochs):
        # set first value
        training_loss = 0
        for j in range(X_tr_nums):
             # random choose data to train
            index = random.sample(range(0,X_tr_nums),1)[0]
            # get prediction values(output)
            out_vals_ = forward_pass(X_tr[index], W, dropout_rate=dropout)
            # update weights value
            W = backward_pass(X_tr[index], Y_tr[index], W, out_vals_, lr=lr, freeze_emb=freeze_emb)
            # calculate the training loss
            training_loss = training_loss + categorical_loss(Y_tr[index],out_vals_['y'])
         # same with training process
        training_loss_history.append(training_loss)

        dev_loss =0
        for j in range(X_dev_nums):
            index = random.sample(range(0,X_dev_nums),1)[0]
            out_vals_ = forward_pass(X_dev[index], W, dropout_rate=dropout)
            dev_loss +=categorical_loss(Y_dev[index],out_vals_['y'])
        # set a condition which the loop will break when the absolute value of difference between
        # the development loss and initial loss value is smaller than tolerance value
        if abs(dev_loss - initial_loss)<tolerance:
            break
        intial_loss = dev_loss
        # store the validation loss value so we can modify our hyperparameter
        validation_loss_history.append(dev_loss)
        if print_progress:
              print('Epoch times: {}, Training loss value: {}, Validation loss: {}'.format(i, trai

    return W, training_loss_history, validation_loss_history
```

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

In [28]:

```
W = network_weights(vocab_size=len(vocab),embedding_dim=500,hidden_dim=[], num_classes=3)

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

W, loss_tr, dev_loss = SGD(X_tr, Y_tr,
                    W,
                    X_dev=X_dev,
                    Y_dev=Y_dev,
                    lr=0.0002,
                    dropout=0.13,
                    freeze_emb=False,
                    tolerance=0.00001,
                    epochs= 100)
```

```
Shape W0 (8931, 500)
Shape W1 (500, 3)
Epoch times: 0, Training loss value: 2600.309010690197, Validation loss: 968.0751727
922072
Epoch times: 1, Training loss value: 2530.8971687832013, Validation loss: 945.302997
2816872
Epoch times: 2, Training loss value: 2467.8321154483438, Validation loss: 923.364356
9192948
Epoch times: 3, Training loss value: 2399.766114026776, Validation loss: 906.5032668
785128
Epoch times: 4, Training loss value: 2331.6717149165283, Validation loss: 889.686223
3610539
Epoch times: 5, Training loss value: 2283.4148988784864, Validation loss: 879.874778
0407047
Epoch times: 6, Training loss value: 2226.8155834496256, Validation loss: 851.069790
8934355
Epoch times: 7, Training loss value: 2179.9088670171795, Validation loss: 833.047456
3090744
Epoch times: 8, Training loss value: 2121.7954351898056, Validation loss: 823.138568
3583885
Epoch times: 9, Training loss value: 2065.134443265823, Validation loss: 806.9236224
213142
Epoch times: 10, Training loss value: 2008.508058945251, Validation loss: 789.280569
5182809
Epoch times: 11, Training loss value: 1965.0828740217414, Validation loss: 774.25772
70086413
Epoch times: 12, Training loss value: 1917.2705221930996, Validation loss: 745.75733
8279313
Epoch times: 13, Training loss value: 1890.049094593021, Validation loss: 736.403607
3048308
Epoch times: 14, Training loss value: 1841.4579491287438, Validation loss: 736.46593
78113323
Epoch times: 15, Training loss value: 1800.9428718793936, Validation loss: 738.82648
63830127
Epoch times: 16, Training loss value: 1790.2868066941728, Validation loss: 699.32688
65095339
Epoch times: 17, Training loss value: 1725.052262303061, Validation loss: 706.107980
3608645
Epoch times: 18, Training loss value: 1704.9876082605467, Validation loss: 680.07257
73185918
Epoch times: 19, Training loss value: 1654.1449216210765, Validation loss: 689.33124
8709145
```

```
Epoch times: 20, Training loss value: 1631.7727089441585, Validation loss: 675.28988
31668164
Epoch times: 21, Training loss value: 1658.1228616802189, Validation loss: 665.00922
11096251
Epoch times: 22, Training loss value: 1612.0581783176606, Validation loss: 661.93435
96938388
Epoch times: 23, Training loss value: 1561.5236050392991, Validation loss: 653.60343
88144478
Epoch times: 24, Training loss value: 1522.2646039372469, Validation loss: 645.42997
45352972
Epoch times: 25, Training loss value: 1544.4789021226172, Validation loss: 632.76966
38399769
Epoch times: 26, Training loss value: 1541.865981958291, Validation loss: 627.180112
8735899
Epoch times: 27, Training loss value: 1517.7700331394528, Validation loss: 617.10875
05590925
Epoch times: 28, Training loss value: 1477.8639739414903, Validation loss: 615.55188
48438768
Epoch times: 29, Training loss value: 1468.5074788328498, Validation loss: 611.62058
60182394
Epoch times: 30, Training loss value: 1434.3649072373776, Validation loss: 652.14250
94334711
Epoch times: 31, Training loss value: 1429.0646401674442, Validation loss: 600.29184
30764652
Epoch times: 32, Training loss value: 1395.6948494561507, Validation loss: 612.02453
605412
Epoch times: 33, Training loss value: 1422.9382464672503, Validation loss: 624.36687
16196024
Epoch times: 34, Training loss value: 1397.6731307021696, Validation loss: 573.52963
1264974
Epoch times: 35, Training loss value: 1381.6924612363805, Validation loss: 561.53558
38772442
Epoch times: 36, Training loss value: 1379.9113016388703, Validation loss: 590.08345
5576326
Epoch times: 37, Training loss value: 1321.0584093917205, Validation loss: 604.64104
81775758
Epoch times: 38, Training loss value: 1322.324938274178, Validation loss: 559.224243
4471517
Epoch times: 39, Training loss value: 1314.0355919646615, Validation loss: 557.04929
6180035
Epoch times: 40, Training loss value: 1341.7502353682498, Validation loss: 539.88865
2144793
Epoch times: 41, Training loss value: 1310.8531503927488, Validation loss: 559.91689
62950104
Epoch times: 42, Training loss value: 1328.505469464734, Validation loss: 547.052730
8448878
Epoch times: 43, Training loss value: 1287.1669360196531, Validation loss: 579.04944
63855987
Epoch times: 44, Training loss value: 1280.6648569599474, Validation loss: 542.82126
4604156
Epoch times: 45, Training loss value: 1260.3792907647928, Validation loss: 576.44235
32853399
Epoch times: 46, Training loss value: 1324.8206473730397, Validation loss: 531.72953
14085467
Epoch times: 47, Training loss value: 1240.5069996192726, Validation loss: 555.59600
61009658
Epoch times: 48, Training loss value: 1271.1146322796803, Validation loss: 497.82315
333411225
Epoch times: 49, Training loss value: 1261.1668867818476, Validation loss: 520.66432
90273901
Epoch times: 50, Training loss value: 1301.813741299205, Validation loss: 539.960151
```

```
1140359
Epoch times: 51, Training loss value: 1230.6250671102166, Validation loss: 506.86754
24366297
Epoch times: 52, Training loss value: 1309.7664958475534, Validation loss: 540.21140
45053182
Epoch times: 53, Training loss value: 1261.7828493564068, Validation loss: 565.89489
489078
Epoch times: 54, Training loss value: 1227.326126863332, Validation loss: 543.954565
1751476
Epoch times: 55, Training loss value: 1195.8647458026076, Validation loss: 526.39525
27949722
Epoch times: 56, Training loss value: 1227.0173543283154, Validation loss: 534.65266
6549347
Epoch times: 57, Training loss value: 1206.8313085195043, Validation loss: 528.70531
12979206
Epoch times: 58, Training loss value: 1290.6599490898082, Validation loss: 572.41409
28451778
Epoch times: 59, Training loss value: 1292.3202898983138, Validation loss: 578.36901
62476422
Epoch times: 60, Training loss value: 1181.1523851362654, Validation loss: 552.08881
04925566
Epoch times: 61, Training loss value: 1152.6290779334854, Validation loss: 520.42142
12085453
Epoch times: 62, Training loss value: 1234.0833464576333, Validation loss: 550.31105
52176688
Epoch times: 63, Training loss value: 1323.310135034301, Validation loss: 536.529619
9132453
Epoch times: 64, Training loss value: 1212.548844042016, Validation loss: 541.216607
0989948
Epoch times: 65, Training loss value: 1134.170209661325, Validation loss: 570.524861
4739071
Epoch times: 66, Training loss value: 1218.2533298718733, Validation loss: 578.77112
20033375
Epoch times: 67, Training loss value: 1251.4166791099078, Validation loss: 593.15807
77697437
Epoch times: 68, Training loss value: 1276.7273967396557, Validation loss: 475.69430
149621434
Epoch times: 69, Training loss value: 1132.8235101397302, Validation loss: 511.27483
9474846
Epoch times: 70, Training loss value: 1233.3172441163406, Validation loss: 578.46650
72156613
Epoch times: 71, Training loss value: 1240.110740345166, Validation loss: 532.446977
3370289
Epoch times: 72, Training loss value: 1298.1569516912368, Validation loss: 541.06199
50546885
Epoch times: 73, Training loss value: 1179.0773917202944, Validation loss: 584.71384
94490871
Epoch times: 74, Training loss value: 1245.2394732990126, Validation loss: 592.92658
08532016
Epoch times: 75, Training loss value: 1234.8383574933393, Validation loss: 530.01074
27626759
Epoch times: 76, Training loss value: 1183.8570507222416, Validation loss: 542.07974
40974571
Epoch times: 77, Training loss value: 1166.5084879747249, Validation loss: 514.04070
02317659
Epoch times: 78, Training loss value: 1251.2298348233837, Validation loss: 500.68740
750829704
Epoch times: 79, Training loss value: 1234.5175364670142, Validation loss: 566.98800
65322048
Epoch times: 80, Training loss value: 1193.9920881052574, Validation loss: 580.19477
23855601
```

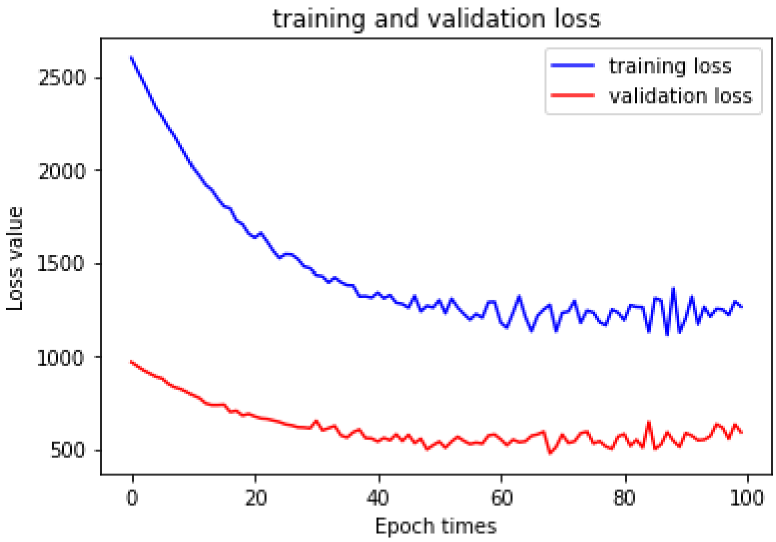
```
Epoch times: 81, Training loss value: 1273.749222526097, Validation loss: 516.950694
3703861
Epoch times: 82, Training loss value: 1265.0458682372002, Validation loss: 549.74543
77955556
Epoch times: 83, Training loss value: 1263.4240465201012, Validation loss: 508.28388
45266586
Epoch times: 84, Training loss value: 1130.5622122876389, Validation loss: 646.20372
6683527
Epoch times: 85, Training loss value: 1311.6022244780577, Validation loss: 499.60845
128247195
Epoch times: 86, Training loss value: 1299.951638564801, Validation loss: 525.534143
3878875
Epoch times: 87, Training loss value: 1113.7475800501418, Validation loss: 590.66459
09267902
Epoch times: 88, Training loss value: 1365.0905486512554, Validation loss: 543.52020
17691995

Epoch times: 89, Training loss value: 1127.5078392062806, Validation loss: 511.89542
53769953
Epoch times: 90, Training loss value: 1204.7479629546583, Validation loss: 585.37046
51991085
Epoch times: 91, Training loss value: 1320.7237110285173, Validation loss: 571.40316
21953288
Epoch times: 92, Training loss value: 1172.70148728041, Validation loss: 546.8774840
716098
Epoch times: 93, Training loss value: 1264.9499986901824, Validation loss: 550.06523
24413106
Epoch times: 94, Training loss value: 1213.0160671546626, Validation loss: 568.81640
63051153
Epoch times: 95, Training loss value: 1255.2794654181328, Validation loss: 633.15249
99234567
Epoch times: 96, Training loss value: 1250.9537235612988, Validation loss: 614.04565
38323189
Epoch times: 97, Training loss value: 1221.7475607572765, Validation loss: 555.19900
01985589
Epoch times: 98, Training loss value: 1294.3318237106016, Validation loss: 631.83119
08959625
Epoch times: 99, Training loss value: 1266.3438375316016, Validation loss: 590.09824
29281423
```

Plot the learning process:

In [29]:

```
plt.plot(range(len(loss_tr)), loss_tr, c = 'blue', label='training loss')
plt.plot(range(len(dev_loss)), dev_loss,c= 'red', label = 'validation loss', )
plt.xlabel('Epoch times')
plt.ylabel('Loss value')
plt.legend()
plt.title('training and validation loss')
plt.show()
```



Compute accuracy, precision, recall and F1-Score:

In [30]:

```
preds_te = [np.argmax(forward_pass(x, W, dropout_rate=0.0)['y']) for x,y in zip(X_te,Y_te)]
print('Accuracy:', accuracy_score(Y_te,preds_te))
print('Precision:', precision_score(Y_te,preds_te,average='macro'))
print('Recall:', recall_score(Y_te,preds_te,average='macro'))
print('F1-Score:', f1_score(Y_te,preds_te,average='macro'))
```

```
Accuracy: 0.8533333333333334
Precision: 0.8626678876678877
Recall: 0.8533333333333334
F1-Score: 0.8521161216942433
```

In [33]:

```
del W;gc.collect()
```

Out[33]:

```
2892
```

Discuss how did you choose model hyperparameters ?

In [37]:

```
results = pd.read_csv("./data_topic/without_embedding.csv")
results
```

Out[37]:

	embedding size	learning_rate	epoch times	training_loss	validation_loss	F1_score
0	300	0.0001	50	1859.363971	726.056014	0.767048
1	300	0.0002	50	1523.284103	609.565260	0.778959
2	300	0.0003	50	1389.563721	563.135827	0.800343
3	300	0.0001	100	1510.786664	624.218801	0.780528
4	300	0.0002	100	1337.724191	575.832935	0.794092
5	300	0.0003	100	1403.003483	617.541038	0.801646
6	500	0.0001	50	1546.848400	622.759730	0.823929
7	500	0.0002	50	1260.862523	520.833905	0.831746
8	500	0.0003	50	1228.350011	507.423733	0.852116
9	500	0.0001	100	1247.861820	554.763552	0.852040
10	500	0.0002	100	1268.311155	588.993825	0.858919
11	500	0.0003	100	1494.924479	704.135099	0.858919

First set the initial value of the learning rate and training times, then conduct model training, and adjust according to the direction of the obtained loss map. If the overall loss value in the image has been declining, it means that we can increase the learning rate and training slightly. Times, but if the loss value has a tendency to become larger at a certain value, it means that we need to reduce the learning rate. Until the obtained loss value reaches a certain stable point, then the parameters we get are relatively good. In this question, we could see the best parameters combination is embedding_size = 500, learning_rate = 0.0002, epoch times = 100, we could get same F1 score when learning_rate equal to 0.0003, but the loss graph is not good.

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.

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:
            print('start laod glove')
            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
    print('laod glove success')
    return w_emb
```

In [30]:

```
w_glove = get_glove_embeddings("glove.840B.300d.zip","glove.840B.300d.txt",word2id)
```

```
start laod glove
laod glove success
```

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,hidden_dim=[], num_classes=3)
W[0] = w_glove

W, loss_tr2, dev_loss2 = SGD(X_tr
                  ,Y_tr
                  ,W
                  ,X_dev=X_dev
                  ,Y_dev=Y_dev
                  ,lr=0.0003
                  ,dropout=0.13
                  ,freeze_emb=True
                  ,tolerance=0.001
                  ,epochs= 100)
```

```
Epoch times: 0, Training loss value: 2467.917977031668, Validation loss: 903.4841431
117256
Epoch times: 1, Training loss value: 2354.42828622795, Validation loss: 860.39868793
52661
Epoch times: 2, Training loss value: 2248.2384252345632, Validation loss: 834.043495
6958086
Epoch times: 3, Training loss value: 2137.3188561763304, Validation loss: 795.595274
2495875
Epoch times: 4, Training loss value: 2064.9585644565473, Validation loss: 762.659046
9633204
Epoch times: 5, Training loss value: 1989.408171736267, Validation loss: 743.7224758
59812
Epoch times: 6, Training loss value: 1900.885831220011, Validation loss: 710.7869113
690413
Epoch times: 7, Training loss value: 1853.769573813848, Validation loss: 691.7333611
874933
Epoch times: 8, Training loss value: 1793.2533923074882, Validation loss: 678.681540
5578245
Epoch times: 9, Training loss value: 1733.4157319108836, Validation loss: 640.332749
6193023
Epoch times: 10, Training loss value: 1703.7300508552582, Validation loss: 623.39900
71024897
Epoch times: 11, Training loss value: 1621.0973730408261, Validation loss: 627.12079
12518945
Epoch times: 12, Training loss value: 1589.9453883365788, Validation loss: 610.53489
37965639
Epoch times: 13, Training loss value: 1555.4308922820405, Validation loss: 591.42689
92970894
Epoch times: 14, Training loss value: 1524.987076291921, Validation loss: 591.972904
9569281
Epoch times: 15, Training loss value: 1480.5979402683697, Validation loss: 582.21253
27566927
Epoch times: 16, Training loss value: 1460.2831002781386, Validation loss: 551.01805
97158575
Epoch times: 17, Training loss value: 1423.8289747073738, Validation loss: 535.92096
31408664
Epoch times: 18, Training loss value: 1401.2762978238006, Validation loss: 544.88711
49001665
Epoch times: 19, Training loss value: 1346.9341783345635, Validation loss: 524.53609
93357729
Epoch times: 20, Training loss value: 1324.258369750955, Validation loss: 532.281454
0264964
Epoch times: 21, Training loss value: 1313.147370685778, Validation loss: 525.697206
```

```
8084519
Epoch times: 22, Training loss value: 1306.663324526487, Validation loss: 504.092398
51488553
Epoch times: 23, Training loss value: 1261.042247583068, Validation loss: 490.943501
96086646
Epoch times: 24, Training loss value: 1248.4988220493522, Validation loss: 492.59710
1714419
Epoch times: 25, Training loss value: 1240.6455966765716, Validation loss: 495.68861
253956345
Epoch times: 26, Training loss value: 1234.8285325323156, Validation loss: 467.11120
385671717
Epoch times: 27, Training loss value: 1216.4068267676869, Validation loss: 482.01616
40711636
Epoch times: 28, Training loss value: 1168.0974898290774, Validation loss: 454.35180
06192899
Epoch times: 29, Training loss value: 1181.044780484498, Validation loss: 485.782857
4465522
Epoch times: 30, Training loss value: 1163.5122412453134, Validation loss: 480.39439
31711258
Epoch times: 31, Training loss value: 1142.2470199859079, Validation loss: 457.38402
56798139
Epoch times: 32, Training loss value: 1163.4014293390605, Validation loss: 462.12254
30311755
Epoch times: 33, Training loss value: 1168.285897551917, Validation loss: 463.281846
59050413
Epoch times: 34, Training loss value: 1141.286774549892, Validation loss: 439.028872
23555865
Epoch times: 35, Training loss value: 1137.049043410486, Validation loss: 476.182653
0593836
Epoch times: 36, Training loss value: 1125.201164061092, Validation loss: 448.808512
5287173
Epoch times: 37, Training loss value: 1127.2859370169826, Validation loss: 419.34915
538550797
Epoch times: 38, Training loss value: 1121.6033695970787, Validation loss: 447.22734
39433933
Epoch times: 39, Training loss value: 1088.7365717760047, Validation loss: 408.64425
553066184
Epoch times: 40, Training loss value: 1059.8914790456595, Validation loss: 380.99560
27332181
Epoch times: 41, Training loss value: 1067.0629553590431, Validation loss: 441.10643
47991247
Epoch times: 42, Training loss value: 1113.50092472889, Validation loss: 409.5807916
093313
Epoch times: 43, Training loss value: 1116.8737745037492, Validation loss: 451.55195
300083244
Epoch times: 44, Training loss value: 1040.7170853199482, Validation loss: 434.10701
500503075
Epoch times: 45, Training loss value: 1007.9711777333245, Validation loss: 436.10330
773310415
Epoch times: 46, Training loss value: 1042.238536597532, Validation loss: 423.000497
64873063
Epoch times: 47, Training loss value: 1047.7335039820816, Validation loss: 413.92876
9938251
Epoch times: 48, Training loss value: 1072.8843428664452, Validation loss: 393.82868
71205731
Epoch times: 49, Training loss value: 1055.5719369050325, Validation loss: 420.52458
76915898
Epoch times: 50, Training loss value: 1023.2690334110162, Validation loss: 424.78409
93813432
Epoch times: 51, Training loss value: 1005.4785776448662, Validation loss: 394.26533
036777926
```

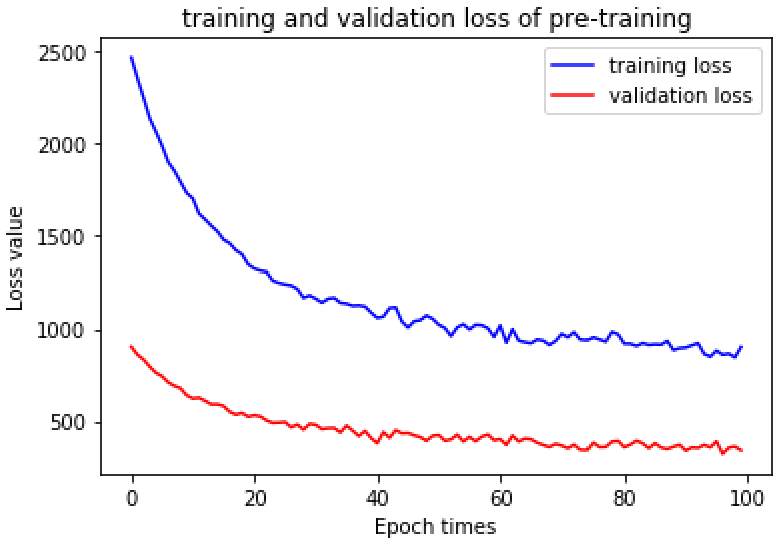
```
Epoch times: 52, Training loss value: 961.6373258337592, Validation loss: 401.010944
85680335
Epoch times: 53, Training loss value: 1008.9563884048554, Validation loss: 427.05062
56048582
Epoch times: 54, Training loss value: 1026.0354126027603, Validation loss: 389.81566
020589685
Epoch times: 55, Training loss value: 997.5765857177264, Validation loss: 415.257977
19140724
Epoch times: 56, Training loss value: 1023.1004633284307, Validation loss: 391.11246
46967486
Epoch times: 57, Training loss value: 1020.5165596602169, Validation loss: 416.27434
62448626
Epoch times: 58, Training loss value: 1004.1785626715109, Validation loss: 427.74655
17560844
Epoch times: 59, Training loss value: 956.8593196827848, Validation loss: 396.584366
8609718
Epoch times: 60, Training loss value: 1019.744865896157, Validation loss: 403.078742
4975895
Epoch times: 61, Training loss value: 926.066332285008, Validation loss: 371.6015571
2643984
Epoch times: 62, Training loss value: 998.9555767744932, Validation loss: 421.814660
73277215
Epoch times: 63, Training loss value: 937.0328630684855, Validation loss: 390.151651
0439466
Epoch times: 64, Training loss value: 929.1136500334674, Validation loss: 405.474895
3866426
Epoch times: 65, Training loss value: 922.6346419008886, Validation loss: 402.680647
5184956
Epoch times: 66, Training loss value: 943.1748746641011, Validation loss: 383.494228
4555159
Epoch times: 67, Training loss value: 937.2023553610208, Validation loss: 370.943691
2793055
Epoch times: 68, Training loss value: 913.6240418204637, Validation loss: 360.264245
1976672
Epoch times: 69, Training loss value: 934.846158491006, Validation loss: 376.8496207
301091
Epoch times: 70, Training loss value: 973.4877672387144, Validation loss: 367.970375
533718
Epoch times: 71, Training loss value: 955.3462931544931, Validation loss: 353.090341
973729
Epoch times: 72, Training loss value: 982.8477755630712, Validation loss: 373.223134
1207363
Epoch times: 73, Training loss value: 942.8764356769465, Validation loss: 345.046468
74402823
Epoch times: 74, Training loss value: 939.4369581856361, Validation loss: 343.084386
39467624
Epoch times: 75, Training loss value: 954.6577258437363, Validation loss: 383.687150
60587004
Epoch times: 76, Training loss value: 943.1912726180376, Validation loss: 359.637036
3219164
Epoch times: 77, Training loss value: 931.333741208064, Validation loss: 360.7764728
4235417
Epoch times: 78, Training loss value: 984.4714201651761, Validation loss: 389.267690
6048325
Epoch times: 79, Training loss value: 971.0369778991409, Validation loss: 393.408808
91125573
Epoch times: 80, Training loss value: 920.3537959029679, Validation loss: 359.099621
1363546
Epoch times: 81, Training loss value: 920.1073431812327, Validation loss: 373.451656
8303137
Epoch times: 82, Training loss value: 907.9606962240828, Validation loss: 394.499148
```

```
56333197
Epoch times: 83, Training loss value: 923.8999974778013, Validation loss: 379.956306
4011565
Epoch times: 84, Training loss value: 913.5188575425943, Validation loss: 353.571577
3135997
Epoch times: 85, Training loss value: 917.2858968363144, Validation loss: 380.677257
3958317
Epoch times: 86, Training loss value: 915.1772037985104, Validation loss: 356.946746
8997344
Epoch times: 87, Training loss value: 935.3651599898536, Validation loss: 349.414038
6078702
Epoch times: 88, Training loss value: 885.7063208406405, Validation loss: 361.814831
4555498


Epoch times: 89, Training loss value: 896.1286106142356, Validation loss: 372.459680
096979
Epoch times: 90, Training loss value: 899.3603377451476, Validation loss: 340.717251
10836064
Epoch times: 91, Training loss value: 911.5781268796326, Validation loss: 358.242345
12782846
Epoch times: 92, Training loss value: 923.049157667369, Validation loss: 354.9606972
690713
Epoch times: 93, Training loss value: 864.8780719106525, Validation loss: 371.857395
3773691
Epoch times: 94, Training loss value: 850.2862776332147, Validation loss: 358.415560
0099787
Epoch times: 95, Training loss value: 882.9749645504908, Validation loss: 391.018986
2475687
Epoch times: 96, Training loss value: 860.4140713200528, Validation loss: 322.775588
9559283
Epoch times: 97, Training loss value: 866.3305496660637, Validation loss: 356.611288
0158469
Epoch times: 98, Training loss value: 846.0914620781159, Validation loss: 363.197529
62109555
Epoch times: 99, Training loss value: 903.0529927617876, Validation loss: 342.043082
8937847
```

In [32]:

```
plt.plot(range(len(loss_tr2)), loss_tr2, c = 'blue', label='training loss')
plt.plot(range(len(dev_loss2)), dev_loss2,c= 'red', label = 'validation loss', )
plt.xlabel('Epoch times')
plt.ylabel('Loss value')
plt.legend()
plt.title('training and validation loss of pre-training')
plt.show()
```



In [33]:

```
preds_te = [np.argmax(forward_pass(x, W, dropout_rate=0.0)['y']) for x,y in zip(X_te,Y_te)]
print('Accuracy:', accuracy_score(Y_te,preds_te))
print('Precision:', precision_score(Y_te,preds_te,average='macro'))
print('Recall:', recall_score(Y_te,preds_te,average='macro'))
print('F1-Score:', f1_score(Y_te,preds_te,average='macro'))
```

```
Accuracy: 0.92
Precision: 0.9220679012345679
Recall: 0.9199999999999999
F1-Score: 0.9203296703296703
```

In [34]:

```
del W;gc.collect()
```

Out[34]:

```
2718
```

Discuss how did you choose model hyperparameters ?

In [35]:

```
results = pd.read_csv("./data_topic/with_embedding.csv")
results
```

Out[35]:

	learning_rate	epochs	training_loss	dev_loss	F1-score
0	0.0001	50	1480.002935	572.505861	0.901070
1	0.0002	50	1184.898596	443.424084	0.907425
2	0.0003	50	1057.130719	396.984397	0.913868
3	0.0001	100	1157.697179	473.807324	0.907425
4	0.0002	100	976.941802	369.611414	0.920330
5	0.0003	100	903.052993	342.043083	0.920330

```
Yes, the F1 socre achieve about 0.9, it's better than without pre-training one
It's a little different, we can't use embedding size = 500, so from the previous tuning
experience, we fix the embedding size to 300, First set the initial value of the learning rate
and training times, then conduct model training, and adjust according to the direction of the
obtained loss map. If the overall loss value in the image has been declining, it means that we
can increase the learning rate and training slightly. Times, but if the loss value has a
tendency to become larger at a certain value, it means that we need to reduce the learning
rate. Until the obtained loss value reaches a certain stable point, then the parameters we get
are relatively good, from the table, we can find the best combination is learaning_rate =
0.0003, epochs = 100, but we can see the F1 score difference among these parameter combination
is small, the smallest value is 0.901070, the best value is 0.920330
```

Extend to support deeper architectures (Bonus)

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.

In [ ]:

```

```

In [ ]:

```
preds_te = [np.argmax(forward_pass(x, W, dropout_rate=0.0)['y']) for x,y in zip(X_te,Y_te)]
print('Accuracy:', accuracy_score(Y_te,preds_te))
print('Precision:', precision_score(Y_te,preds_te,average='macro'))
print('Recall:', recall_score(Y_te,preds_te,average='macro'))
print('F1-Score:', f1_score(Y_te,preds_te,average='macro'))
```

Full Results

Add your final results here:

Model	Precision	Recall	F1-Score	Accuracy
Average Embedding	0.8721804511278196	0.86	0.8589193825042881	0.86
Average Embedding (Pre-trained)	0.92	0.9220679012345679	0.9199999999999999	0.9203296703296703
Average Embedding (Pre-trained) + X hidden layers (BONUS)				

In [ ]:

```
```