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

Instructor: Nikos Aletras

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

For that purpose, you will implement:

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

Data

The data you will use for the task is a subset of the [AG News Corpus (http://groups.di.unipi.it/~gulli/AG_corpus_of_news_articles.html)](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, recall_score, f1_score
import random
from time import localtime, strftime
from scipy.stats import spearmanr,pearsonr
import zipfile
from prettytable import PrettyTable
import gc
import copy
import time
# 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]:

```
start_time = time.time()
training_data = pd.read_csv("data_topic/train.csv")
development_data = pd.read_csv("data_topic/dev.csv")
test_data = pd.read_csv("data_topic/test.csv")
```

In [3]:

```
train_content = training_data.iloc[:,1].to_list()
development_content = development_data.iloc[:,1].to_list()
test_content = test_data.iloc[:,1].to_list()
train_label = training_data.iloc[:,0].to_numpy()
development_label = development_data.iloc[:,0].to_numpy()
test_label = test_data.iloc[:,0].to_numpy()
```

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_w

    tokenRE = re.compile(token_pattern)

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

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

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

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

        # ignore unigrams
        if n==1: continue

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

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

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

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

    return x
```

Create a vocabulary of n-grams

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

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

and returns:

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

In [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()

    # iterate through each raw text
    for x in X_raw:

        x_ngram = extract_ngrams(x, ngram_range=ngram_range, token_pattern=token_pattern, s

        #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 elements with doc frequency > minimum term freq (keep_topN)
    # if use keep_topN as the most frequent N words to keep,
    # you don't know which word to choose if N and N+1 words have the same frequency.
    # so I choose to keep words which appear more than N times.
    if keep_topN>0:
        vocab = set([w for w in ngram_counts if ngram_counts[w]>=keep_topN])


    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, tf = get_vocab(X_raw=train_content, min_df=0, keep_topN=0, stop_words=stop_words
```

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

In [8]:

```
vocab_to_id = {}
id_to_vocab = {}
i = 0
for word in vocab:
    vocab_to_id[word]=i
    id_to_vocab[i] = word
    i += 1
```

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]:

```
processed_train=[]
processed_development=[]
processed_test=[]
for i in range(len(train_content)):
    processed_train.append(extract_ngrams(x_raw=train_content[i], ngram_range=(1,1),
                                          stop_words=stop_words, vocab=vocab))
for i in range(len(development_content)):
    processed_development.append(extract_ngrams(x_raw=development_content[i], ngram_range=(
                                                stop_words=stop_words, vocab=vocab))
for i in range(len(test_content)):
    processed_test.append(extract_ngrams(x_raw=test_content[i], ngram_range=(1,1),
                                         stop_words=stop_words, vocab=vocab))
```

Then convert them into lists of indices in the vocabulary:

In [10]:

```
# Here I use deepcopy because I may use the processed train later
train_indices = copy.deepcopy(processed_train)
development_indices = copy.deepcopy(processed_development)
test_indices = copy.deepcopy(processed_test)
for i in range(len(train_indices)):
    for j in range(len(train_indices[i])):
        train_indices[i][j] = vocab_to_id[train_indices[i][j]]
for i in range(len(development_indices)):
    for j in range(len(development_indices[i])):
        development_indices[i][j] = vocab_to_id[development_indices[i][j]]
for i in range(len(test_indices)):
    for j in range(len(test_indices[i])):
        test_indices[i][j] = vocab_to_id[test_indices[i][j]]
```

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

In [11]:

```
# change the labels from [1,2,3] to [0,1,2]
train_label = train_label-1
development_label = development_label-1
test_label = test_label-1
```

Network Architecture

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

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

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

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

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

Finally the hidden layer is passed to the output layer:

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

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

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

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

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

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

Network Training

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

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

and returns:

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

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

In [12]:

```
def network_weights(vocab_size=1000, embedding_dim=300,
                    hidden_dim=[], num_classes=3, init_val = 0.5):
    W = dict()
    W[0] = np.random.uniform(-init_val, init_val, (vocab_size, embedding_dim))
    for i in range(1, len(hidden_dim)+1):
        if i == 1:
            W[i] = np.random.uniform(-init_val, init_val, (embedding_dim, hidden_dim[i-1]))
        else:
            W[i] = np.random.uniform(-init_val, init_val, (hidden_dim[i-2], hidden_dim[i-1]

    # get the W between y and the last hidden layer
    if len(hidden_dim)>0:
        W[len(hidden_dim)+1] = np.random.uniform(-init_val, init_val, (hidden_dim[len(hidde
    # if there is no hidden layer, W[1] is the weight between embedding layer and y
    else:
        W[1] = np.random.uniform(-init_val, init_val, (embedding_dim, num_classes))
    return W

# Embedding layer = X * W[0]
# hidden layer 1 = embedding layer * W[1]
# hidden layer 2 = hidden layer 1 * W[2]
# ......
# num_classes = hidden layer n * W[n+1]
```

In [13]:

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

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

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

In [14]:

```
def softmax(z):
    sig = np.exp(z)/sum(np.exp(z))

    return sig

# exp() all elements in z and divided by the sum of exp(z)
```

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 [15]:

```
def categorical_loss(y, y_preds):
    l = -y * np.log(max(y_preds))
    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 [16]:

```
def relu(z):
    z_copy = z.copy()
    # return 0 if the value of z is less than 0
    a = np.maximum(z_copy, 0)
    return a

def relu_derivative(z):
    z_copy = z.copy()
    dz = np.empty(len(z_copy))
    for i in range(len(z_copy)):
        if z_copy[i] > 0:
            dz[i] = 1
        else:
            dz[i] = 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 [17]:

```
def dropout_mask(size, dropout_rate):
    dropout_vec = np.ones(size)
    indices = np.random.choice(np.arange(size), replace=False, size=int(size * dropout_rate
    dropout_vec[indices] = 0
    return dropout_vec
```

In [18]:

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

```
[1. 1. 1. 1. 1. 1. 1. 1. 0. 0.]
[1. 0. 1. 1. 1. 1. 1. 0. 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 [19]:

```
def forward_pass(x, W, dropout_rate=0.2):
    out_vals = {}
    h_vecs = []
    a_vecs = []
    dropout_vecs = []
    # first, compute for the embedding layer
    vec = [W[0][indices] for indices in x]
    embedding_layer = np.sum(vec,axis = 0)/len(x)
    h_vecs.append(embedding_layer)
    a0 = relu(embedding_layer)
    a_vecs.append(a0)
    dropout_vec0 = dropout_mask(len(a0), dropout_rate)
    dropout_vecs.append(dropout_vec0)
    after_dropout = embedding_layer*dropout_vec0

    # then, compute the other hidden layers
    for i in range(1,len(W)-1):
        h = np.dot(after_dropout, W[i])
        a = relu(h)
        dropout_vec = dropout_mask(len(a), dropout_rate)
        after_dropout = a*dropout_vec

        h_vecs.append(h)
        a_vecs.append(a)
        dropout_vecs.append(dropout_vec)

    # compute the prediction
    prediction = softmax(np.dot(after_dropout,W[len(W)-1]))
    # set the out_vals
    out_vals['h'] = h_vecs
    out_vals['a'] = a_vecs
    out_vals['dropout_vecs'] = dropout_vecs
    out_vals['prediction'] = prediction

    return out_vals
```

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

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

and returns:

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

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

In [20]:

```
def backward_pass(x, y, W, out_vals, lr=0.001, freeze_emb=False):
    # first compute the W between output and last hidden layer
    # Use the last W to set the length of array of labels
    true_labels = np.zeros(W[len(W)-1].shape[1])
    # set the true category to 1
    true_labels[y] = 1
    # compute the difference between true labels and prediction
    gradient = (out_vals['prediction'] - true_labels).reshape(1, -1) # (n, ) reshape to (1,
    last_layer = (out_vals['a'][-1]*out_vals['dropout vecs'][-1]).reshape(-1, 1) # (m, ) re
    dw = np.dot(last_layer, gradient) # mxn
    # change w
    W[len(W)-1] = W[len(W)-1] - lr*dw # mxn
    gradient = (np.dot(W[len(W)-1],gradient.T)) * out_vals['dropout vecs'][len(W)-2].reshap

    for i in range(1, len(W)-1):
        gradient = gradient * relu_derivative(out_vals['h'][len(W)-i-1]).reshape(-1,1) # (m
        last_layer = (out_vals['a'][len(W)-i-2]*out_vals['dropout vecs'][len(W)-i-2]).resha
        dw = np.dot(last_layer, gradient.T)
        # change w
        W[len(W)-i-1] = W[len(W)-i-1] - lr*dw
        gradient = np.dot(W[len(W)-i-1], gradient) * out_vals['dropout vecs'][len(W)-i-2].r

    # decide whether to change W[0] or not
    if freeze_emb == False:
        gradient = gradient * relu_derivative(out_vals['h'][0]).reshape(-1,1)
        W[0][x] = W[0][x] - lr*(gradient.reshape(-1,))

    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 [21]:

```
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):
    training_loss_history = []
    validation_loss_history = []
    index = np.arange(len(X_tr))
    for i in range(epochs):
        random.Random(32*i).shuffle(index)

        for indices in index:
            out_vals = forward_pass(x=X_tr[indices], W=W, dropout_rate=dropout)
            # update w
            W = backward_pass(x=X_tr[indices], y=Y_tr[indices], W=W, out_vals=out_vals, lr=

        training_loss = 0
        for j in range(len(X_tr)):
            y_pred_train = forward_pass(x=X_tr[j], W=W, dropout_rate=dropout)['prediction']
            training_loss = categorical_loss(Y_tr[j], y_pred_train) + training_loss
        training_loss_history.append(training_loss/len(X_tr))

        validation_loss = 0
        for k in range(len(X_dev)):
            y_pred_dev = forward_pass(x=X_dev[k], W=W, dropout_rate=dropout)['prediction']
            validation_loss = categorical_loss(Y_dev[k], y_pred_dev) + validation_loss
        validation_loss_history.append(validation_loss/len(X_dev))

        if i>0:
            if abs(validation_loss_history[i-1]-validation_loss_history[i])<tolerance:
                break
    if print_progress==True:
        print('train loss: ', training_loss_history)
        print('------------------------------------------------')
        print('val loss: ', validation_loss_history)

    return W, training_loss_history, validation_loss_history
```

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

In [22]:

```
W = network_weights(vocab_size=len(vocab),embedding_dim=50,
                    hidden_dim=[], num_classes=3)

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

```
Shape W0 (81707, 50)
Shape W1 (50, 3)
```

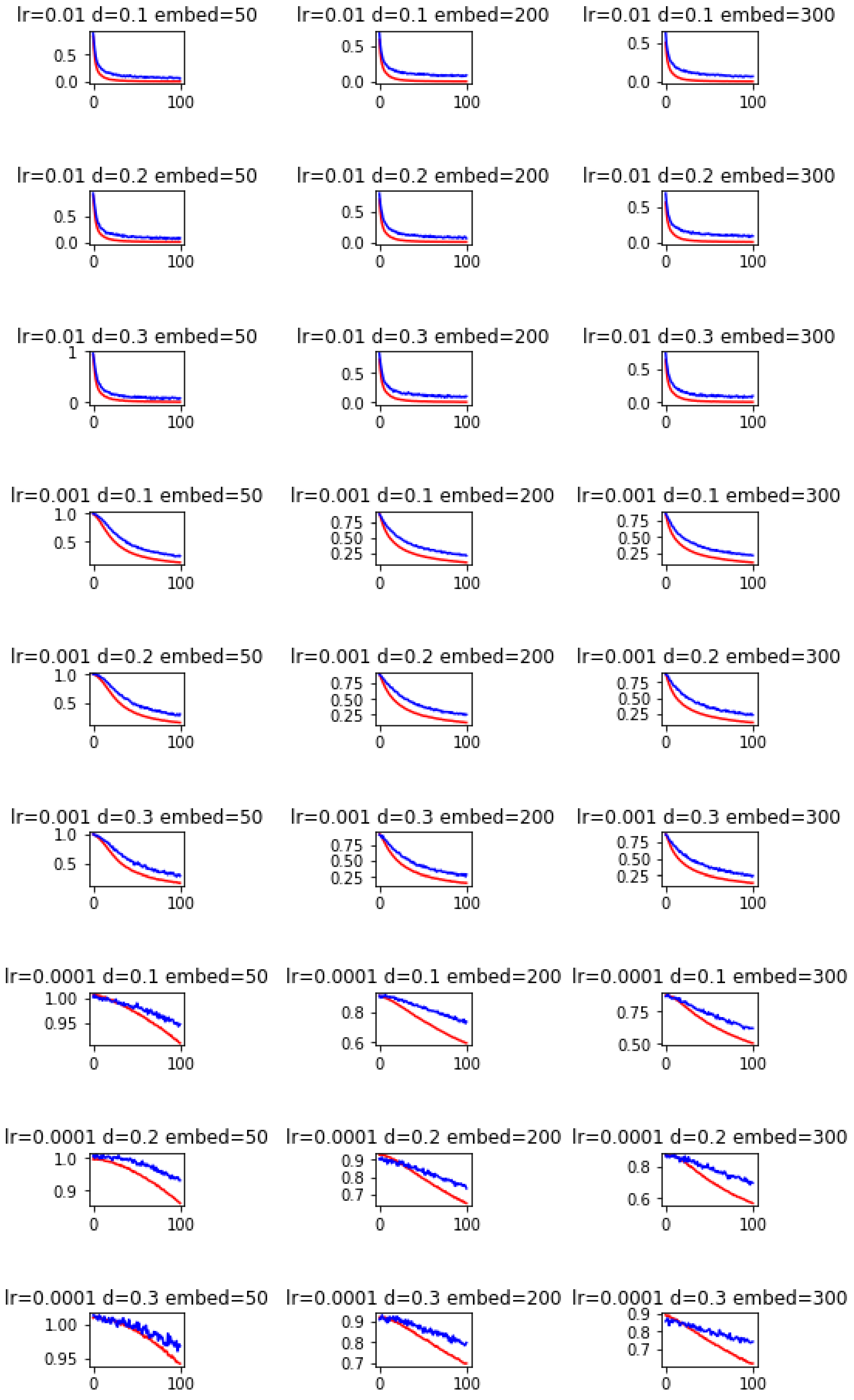
Discuss how did you choose model hyperparameters ?

In [23]:

```
tune1_start = time.time()
lr = [0.01, 0.001, 0.0001]
dropout = [0.1, 0.2, 0.3]
embedding_dim = [50, 200, 300]
# create the table showing performance of each model. Here I use accuracy as the performanc
table_lr0 = PrettyTable(["lr = {}".format(lr[0]), "dropout = {}".format(dropout[0]), "dropo
            , "dropout = {}".format(dropout[2])])
table_lr1 = PrettyTable(["lr = {}".format(lr[1]), "dropout = {}".format(dropout[0]), "dropo
            , "dropout = {}".format(dropout[2])])
table_lr2 = PrettyTable(["lr = {}".format(lr[2]), "dropout = {}".format(dropout[0]), "dropo
            , "dropout = {}".format(dropout[2])])
acc_history = []

z=1

for i in range(len(lr)):
    for j in range(len(dropout)):
        for k in range(len(embedding_dim)):
            # train model with different parameters
            W = network_weights(vocab_size=len(vocab),embedding_dim=embedding_dim[k],
                    hidden_dim=[], num_classes=3)

            W, loss_tr, dev_loss = SGD(train_indices, train_label,
                                        W,
                                        X_dev=development_indices,
                                        Y_dev=development_label,
                                        lr=lr[i],
                                        dropout=dropout[j],
                                        freeze_emb=False,
                                        tolerance=0.0000001,
                                        epochs=100, print_progress=False)
            # choose the best model by selecting the highest accuracy on the dev set
            preds_te = [np.argmax(forward_pass(x, W, dropout_rate=0.0)['prediction'])
            for x,y in zip(development_indices,development_label)]
            acc_history.append(accuracy_score(development_label,preds_te))
            plt.subplot(9,3,z)
            plt.plot(loss_tr, 'r')
            plt.plot(dev_loss, 'b')
            plt.title('lr={}'.format(lr[i])+' d={}'.format(dropout[j])+' embed={}'.format(e

            z+=1

    if i == 0:
        table_lr0.add_row(["embedding_dim={}".format(embedding_dim[0]), acc_history[i*9+0],
        table_lr0.add_row(["embedding_dim={}".format(embedding_dim[1]), acc_history[i*9+1],
        table_lr0.add_row(["embedding_dim={}".format(embedding_dim[2]), acc_history[i*9+2],
    if i == 1:
        table_lr1.add_row(["embedding_dim={}".format(embedding_dim[0]), acc_history[i*9+0],
        table_lr1.add_row(["embedding_dim={}".format(embedding_dim[1]), acc_history[i*9+1],
        table_lr1.add_row(["embedding_dim={}".format(embedding_dim[2]), acc_history[i*9+2],
    if i == 2:
        table_lr2.add_row(["embedding_dim={}".format(embedding_dim[0]), acc_history[i*9+0],
        table_lr2.add_row(["embedding_dim={}".format(embedding_dim[1]), acc_history[i*9+1],
        table_lr2.add_row(["embedding_dim={}".format(embedding_dim[2]), acc_history[i*9+2],

result = max(acc_history)
print('max accuracy:', result)
index_of_result = acc_history.index(result)
lr_index = (index_of_result)//9
```

```
dropout_index = (index_of_result%9)//3
embedding_dim_index = (index_of_result%3)
print('lr: ', lr[lr_index])
print('dropout: ', dropout[dropout_index])
print('hidden_dim: ', embedding_dim[embedding_dim_index])
print(table_lr0)
print(table_lr1)
print(table_lr2)
plt.subplots_adjust(left=6,
                    bottom=2,
                    right=7,
                    top=5,
                    wspace=2,
                    hspace=2)
plt.show()
tune1_end = time.time()
```

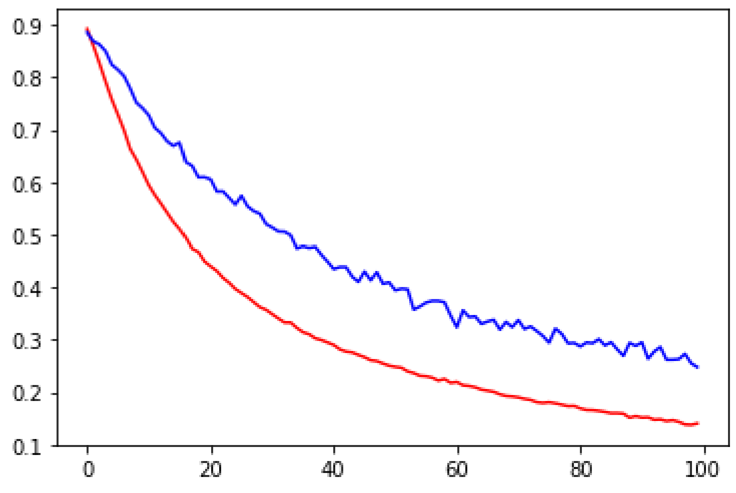
```
max accuracy: 0.9060402684563759
lr:  0.001
dropout:  0.3
hidden_dim:  300
+-------------------+--------------------+--------------------+-------------
-------+
|     lr = 0.01     |   dropout = 0.1    |   dropout = 0.2    |   dropout =
0.3    |
+-------------------+--------------------+--------------------+-------------
-------+
|  embedding_dim=50 | 0.8657718120805369 |  0.87248322147651  | 0.8926174496
644296 |
| embedding_dim=200 | 0.8657718120805369 |  0.87248322147651  |  0.872483221
47651  |
| embedding_dim=300 | 0.8657718120805369 | 0.8859060402684564 | 0.8859060402
684564 |
+-------------------+--------------------+--------------------+-------------
-------+
+-------------------+--------------------+--------------------+-------------
-------+
|     lr = 0.001    |   dropout = 0.1    |   dropout = 0.2    |   dropout =
0.3    |
+-------------------+--------------------+--------------------+-------------
-------+
|  embedding_dim=50 | 0.8926174496644296 | 0.8926174496644296 | 0.8926174496
644296 |
| embedding_dim=200 | 0.8926174496644296 | 0.8993288590604027 | 0.8993288590
604027 |
| embedding_dim=300 | 0.8926174496644296 | 0.8993288590604027 | 0.9060402684
563759 |
+-------------------+--------------------+--------------------+-------------
-------+
+-------------------+--------------------+--------------------+-------------
-------+
|    lr = 0.0001    |   dropout = 0.1    |   dropout = 0.2    |   dropout =
0.3    |
+-------------------+--------------------+--------------------+-------------
-------+
|  embedding_dim=50 | 0.6040268456375839 | 0.6778523489932886 | 0.6174496644
295302 |
| embedding_dim=200 | 0.7718120805369127 | 0.7785234899328859 | 0.7114093959
731543 |
| embedding_dim=300 | 0.8120805369127517 | 0.7583892617449665 | 0.7046979865
771812 |
+-------------------+--------------------+--------------------+-------------
-------+
```



In [24]:

```
# After finding the best parameters, apply them to test data
W = network_weights(vocab_size=len(vocab),embedding_dim=embedding_dim[embedding_dim_index],
                    hidden_dim=[], num_classes=3)

W, loss_tr, dev_loss = SGD(train_indices, train_label,
                            W,
                            X_dev=development_indices,
                            Y_dev=development_label,
                            lr=lr[lr_index],
                            dropout=dropout[dropout_index],
                            freeze_emb=False,
                            tolerance=0.0000001,
                            epochs=100)

# plot the graph of training loss and validation loss
plt.plot(loss_tr, 'r')
plt.plot(dev_loss, 'b')

# get the accuracy, precision, etc. of the best model
preds_te = [np.argmax(forward_pass(x, W, dropout_rate=0.0)['prediction'])
            for x,y in zip(test_indices,test_label)]

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

```
train loss:  [0.8920452445926531, 0.8605016614799051, 0.82664638004869,
0.7915948023586815, 0.7578050784811154, 0.7286539883664198, 0.69880782150
84607, 0.6631233705671401, 0.6428682465047336, 0.6192026450304429, 0.5943
316521729303, 0.5752673579102366, 0.558812336784006, 0.5414735647282138,
0.5241788664843456, 0.5102943619983378, 0.4938817254572439, 0.47347672351
835685, 0.46723971941115766, 0.4492833312710699, 0.43986601287931737, 0.4
3117632704284065, 0.41807607048049855, 0.408741833937936, 0.3969988960469
393, 0.38913191447177914, 0.38149195424030824, 0.3717503330808036, 0.3622
8776920518935, 0.35719868219819584, 0.34886402369977737, 0.34066245869552
686, 0.3329649925025638, 0.3329151128163287, 0.3226651431551839, 0.314416
6147554223, 0.31054734359113945, 0.30276837497329157, 0.2995121632946955,
0.29481893678135634, 0.2901141703460635, 0.28168235945588005, 0.277482119
1801287, 0.27587226113862345, 0.27106379253858076, 0.266842765260662, 0.2
6096677122575596, 0.2593077958643308, 0.2546643894113693, 0.2505831431300
861, 0.24827981768796237, 0.2466457949503736, 0.23967857750133567, 0.2364
554366235861, 0.2312951819766789, 0.23030236304689936, 0.227703741607023
9, 0.22208230625158135, 0.225165565463111, 0.21770907246513796, 0.2194218
7409970662, 0.21313626338811098, 0.211823520688739, 0.20929692019529247,
0.2048931809202036, 0.20285943550705662, 0.20063173384611416, 0.195915335
33389288, 0.19281994462842117, 0.19183983640083427, 0.1900956994116153,
0.18717310935206394, 0.18527271831001424, 0.18086814602722265, 0.17947365
493581188, 0.18081511595964966, 0.17878298448785815, 0.17604271040271416,
0.17338585675942528, 0.17376126460915675, 0.16895683865149994, 0.16602476
2536413, 0.16571020217886795, 0.1643033496121854, 0.16229007571859655, 0.
15988591629670093, 0.1599848476149901, 0.15898977554460283, 0.15132321339
935698, 0.15428130434309498, 0.15152383595418722, 0.15205877816434576, 0.
14741664365956164, 0.14814902021769527, 0.14490224866196738, 0.1465417703
9620472, 0.1433879767080457, 0.13841580685575086, 0.13766046952776864, 0.
1403153191063666]
------------------------------------------------
val loss:  [0.8854140054357971, 0.8687069713406, 0.8625445195630413, 0.84
97063020764114, 0.8241039554691972, 0.8142716480180267, 0.802238868941816
2, 0.7786533876861893, 0.7515920285037013, 0.7410770424967859, 0.72768242
42508511, 0.703556895449062, 0.6932874838030808, 0.6777605899548356, 0.66
91355518092232, 0.6752724069309096, 0.6378788370068686, 0.63079792836565
8, 0.6090429265063215, 0.6093211772044745, 0.6051100857132191, 0.58190575
12070654, 0.5821377020942442, 0.5697492046903903, 0.5570306058586602, 0.5
73816420488934, 0.553516126069254 5, 0.5445480996598474, 0.539512529295116
4, 0.5199940204052814, 0.5133963910282545, 0.5058352617312806, 0.50542607
7867892, 0.4987388236893927, 0.4739286633319592, 0.4780823089058388, 0.47
47661434327885, 0.47701105432174773, 0.4617713676584218, 0.44892224264063
59, 0.434743049797285, 0.438663470557124, 0.43862246179311065, 0.42030187
03894249, 0.4104202304355298, 0.43016324621541735, 0.4133080779695201, 0.
42881204589330035, 0.40679741097683664, 0.4095012289365141, 0.39425631504
890696, 0.3973551847886719, 0.39639734257101733, 0.35749295672604253, 0.3
635047760346386, 0.37091793823687735, 0.37415893472377293, 0.374096091802
3, 0.37161943854919244, 0.3459830097444884, 0.32375107012685317, 0.356203
98331679937, 0.3433776361236257, 0.34458699011492383, 0.3302896215467306
4, 0.33483136999818297, 0.3381382549013331, 0.31932970621611917, 0.334046
25025921025, 0.32407324742800414, 0.33757491405479795, 0.320514183137838,
0.3254843753705545, 0.31653668132275314, 0.3072271444445206, 0.2943710272
6943964, 0.3210677324999421, 0.3108388483069677, 0.2931706512831253, 0.29
395862451893423, 0.28723275747958465, 0.29477053402286063, 0.293397833069
76047, 0.3011913493581923, 0.289238084637265, 0.2952894324959022, 0.28162
636601930624, 0.2693081261715536, 0.29403527848620753, 0.288356756305598
9, 0.2951321900655637, 0.26367044489901054, 0.27750256272453716, 0.286179
89800981836, 0.2619174235543469, 0.26200091448734003, 0.2629290639885243,
0.27309239997841217, 0.25530236173336635, 0.24779580063789286]
Accuracy: 0.8498331479421579
Precision: 0.8527593830614076
Recall: 0.8498253437383871
F1-Score: 0.8494664856906962
```



The model is neither overfitting nor underfitting. The train loss doesn't bounce back means this model doesn't overfit and the loss is reducing as the epochs getting bigger means this model doesn't underfit. Though the difference between train and validation loss is larger than the other two models, it is still acceptable. Also, the accuracy is about 85%, which is great.

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 [25]:

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

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

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

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

In [26]:

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

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 [27]:

```
w_pretrained = network_weights(vocab_size=len(vocab),embedding_dim=w_glove.shape[1],hidden_
w_pretrained[0] = w_glove

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

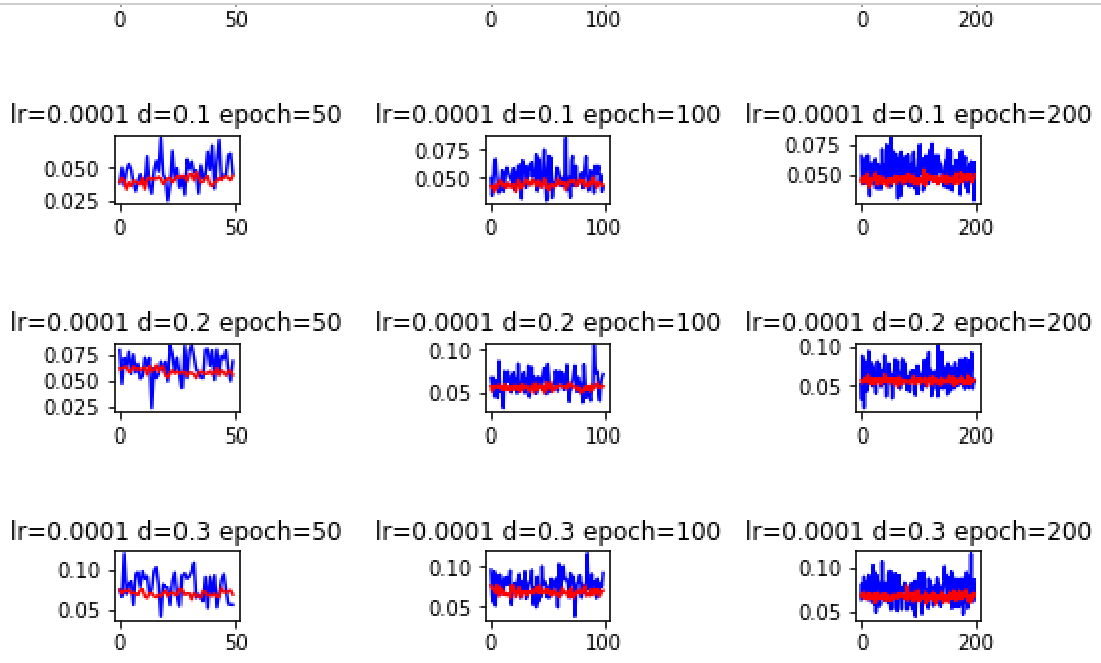
```
Shape w_pretrained0 (81707, 300)
Shape w_pretrained1 (300, 3)
```

Discuss how did you choose model hyperparameters ?

In [28]:

```
tune2_start = time.time()
lr = [0.01, 0.001, 0.0001]
dropout = [0.1, 0.2, 0.3]
epochs = [50, 100, 200]
# create the table showing performance of each model. Here I use accuracy as the performanc
table_lr0 = PrettyTable(["lr = {}".format(lr[0]), "dropout = {}".format(dropout[0]), "dropo
            , "dropout = {}".format(dropout[2])])
table_lr1 = PrettyTable(["lr = {}".format(lr[1]), "dropout = {}".format(dropout[0]), "dropo
            , "dropout = {}".format(dropout[2])])
table_lr2 = PrettyTable(["lr = {}".format(lr[2]), "dropout = {}".format(dropout[0]), "dropo
            , "dropout = {}".format(dropout[2])])
acc_history = []

z=1
w_pretrained = network_weights(vocab_size=len(vocab),embedding_dim=w_glove.shape[1],
        hidden_dim=[], num_classes=3)
w_pretrained[0] = w_glove
for i in range(len(lr)):
    for j in range(len(dropout)):
        for k in range(len(epochs)):
            # train model with different parameters
            W, loss_tr, dev_loss = SGD(train_indices, train_label,
                                        w_pretrained,
                                        X_dev=development_indices,
                                        Y_dev=development_label,
                                        lr=lr[i],
                                        dropout=dropout[j],
                                        freeze_emb=True,
                                        tolerance=0.0000001,
                                        epochs=epochs[k], print_progress=False)
            # choose the best model by selecting the highest accuracy on the dev set
            preds_te = [np.argmax(forward_pass(x, W, dropout_rate=0.0)['prediction'])
            for x,y in zip(development_indices,development_label)]
            acc_history.append(accuracy_score(development_label,preds_te))
            plt.subplot(9,3,z)
            plt.plot(dev_loss, 'b')
            plt.plot(loss_tr, 'r')
            plt.title('lr={}'.format(lr[i])+' d={}'.format(dropout[j])+' epoch={}'.format(e

            z+=1

    if i == 0:
        table_lr0.add_row(["epochs = {}".format(epochs[0]), acc_history[i*9+0], acc_history
        table_lr0.add_row(["epochs = {}".format(epochs[1]), acc_history[i*9+1], acc_history
        table_lr0.add_row(["epochs = {}".format(epochs[2]), acc_history[i*9+2], acc_history
    if i == 1:
        table_lr1.add_row(["epochs = {}".format(epochs[0]), acc_history[i*9+0], acc_history
        table_lr1.add_row(["epochs = {}".format(epochs[1]), acc_history[i*9+1], acc_history
        table_lr1.add_row(["epochs = {}".format(epochs[2]), acc_history[i*9+2], acc_history
    if i == 2:
        table_lr2.add_row(["epochs = {}".format(epochs[0]), acc_history[i*9+0], acc_history
        table_lr2.add_row(["epochs = {}".format(epochs[1]), acc_history[i*9+1], acc_history
        table_lr2.add_row(["epochs = {}".format(epochs[2]), acc_history[i*9+2], acc_history

result = max(acc_history)
print('max accuracy:', result)
index_of_result = acc_history.index(result)
lr_index = (index_of_result)//9
dropout_index = (index_of_result%9)//3
```

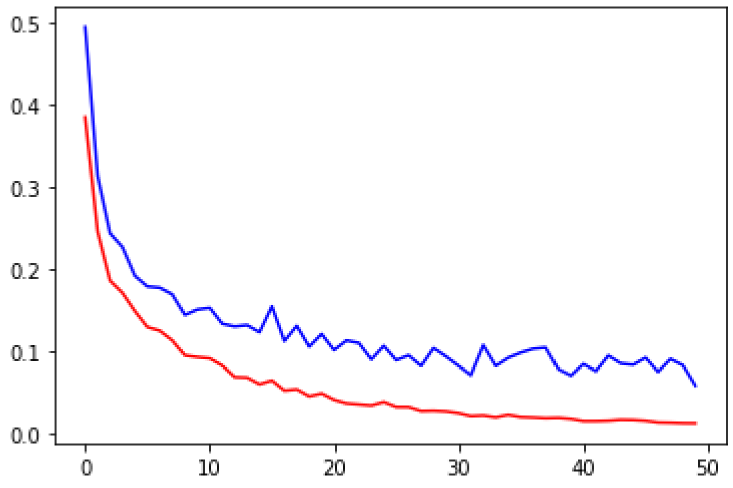
```
epochs_index = (index_of_result%3)
print('lr: ', lr[lr_index])
print('dropout: ', dropout[dropout_index])
print('epochs: ', epochs[epochs_index])
print(table_lr0)
print(table_lr1)
print(table_lr2)
plt.subplots_adjust(left=6,
                    bottom=2,
                    right=7,
                    top=5,
                    wspace=2,
                    hspace=2)
plt.show()
tune2_end = time.time()
```



In [29]:

```
# After finding the best parameters, apply them to test data
w_pretrained = network_weights(vocab_size=len(vocab),embedding_dim=w_glove.shape[1],
                    hidden_dim=[], num_classes=3)
w_pretrained[0] = w_glove

W, loss_tr, dev_loss = SGD(train_indices, train_label,
                            w_pretrained,
                            X_dev=development_indices,
                            Y_dev=development_label,
                            lr=lr[lr_index],
                            dropout=dropout[dropout_index],
                            freeze_emb=False,
                            tolerance=0.0000001,
                            epochs=epochs[epochs_index])

# plot the graph of training loss and validation loss
plt.plot(loss_tr, 'r')
plt.plot(dev_loss, 'b')

# get the accuracy, precision, etc. of the best model
preds_te = [np.argmax(forward_pass(x, W, dropout_rate=0.0)['prediction'])
            for x,y in zip(test_indices,test_label)]

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

```
train loss:  [0.3842252908298302, 0.2456647403358841 5, 0.185743848401836
9, 0.17076008461050707, 0.14850151627988764, 0.12911409328677328, 0.12466
449588741454, 0.11290790136479532, 0.09478581739607418, 0.09265165106038
7, 0.09120234375680708, 0.0824148276727453, 0.0683526960646884, 0.0677348
7873316814, 0.05963952674124515, 0.06418548386930269, 0.05195342377666696
5, 0.05341319716096213, 0.04496882070233448, 0.04840079887111891, 0.04071
8724738784594, 0.036431302365252584, 0.03518101885172625, 0.0339249947532
0996, 0.03808588630694068, 0.03212815377962983, 0.032018231058792423, 0.0
27220664491122575, 0.027548464618073767, 0.02677808118323548, 0.024676643
891527045, 0.02106821214809842 2, 0.021808024071232626, 0.019550568621842 6
3, 0.02261023514616903 5, 0.019889614128205 72, 0.019331793276342 64, 0.0185
15038045906054, 0.018897113224599763, 0.017595792182966728, 0.01493301633
4175217, 0.014952232388406116, 0.015376124967129203, 0.016543376538346 93,
0.016278415633142952, 0.015292802006901042, 0.013211137117678871, 0.01290
9879888981766, 0.01240288302446815, 0.012246476221955491]
------------------------------------------------
val loss:  [0.4943275164474107 4, 0.313577030758063 5, 0.243458304026416 13,
0.2265299091573882, 0.191523334729166 2, 0.17854990128275258, 0.1770620498
048544, 0.16896158924574364, 0.14380608651960822, 0.15043768783544836, 0.
1524733252619929 8, 0.13316016466587066, 0.12979768598876 7, 0.131487963702
73313, 0.1228083237383601 5, 0.15436491930579208, 0.11200822293455592, 0.1
3061457244008284, 0.10525753998710179, 0.12067441625836246, 0.10114016007
638149, 0.11285023171503221, 0.10978542808061925, 0.0897732611628023, 0.1
0635639662074205, 0.0889326127828765 4, 0.09490314478312797, 0.08198445596
438267, 0.10365799002909461, 0.09352378211090877, 0.08221199156890799, 0.
07064398688175527, 0.10727529566734884, 0.08190681140788726, 0.0918891926
4248366, 0.09802318651954174, 0.10276738050433801, 0.10434807804105806,
0.07775392551525709, 0.07014878676085728, 0.08443943386470207, 0.07547583
71733071, 0.09461258519215605, 0.085118017675114 7, 0.08343704550449 16, 0.
09206702470337762, 0.0746172272105514, 0.0905213913974253, 0.082800170764
74812, 0.0579648591450403]
Accuracy: 0.8720800889877642
Precision: 0.8733180409890507
Recall: 0.8720549981419546
F1-Score: 0.8721743420278848
```



This model is neither overfitting nor underfitting because of the same reason in the first model. Also, the validation loss (the blue line) is lower than 0.1, which means it performs real good on validation set. Moreover, the accuracy of the model is about 87.2%. To conclude, this is an accurate model without overfitting and underfitting.

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 [30]:

```
w_extended = network_weights(vocab_size=len(vocab),embedding_dim=w_glove.shape[1],hidden_di
w_extended[0] = w_glove

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

```
Shape w_extended0 (81707, 300)
Shape w_extended1 (300, 200)
Shape w_extended2 (200, 3)
```

Discuss how did you choose model hyperparameters ?

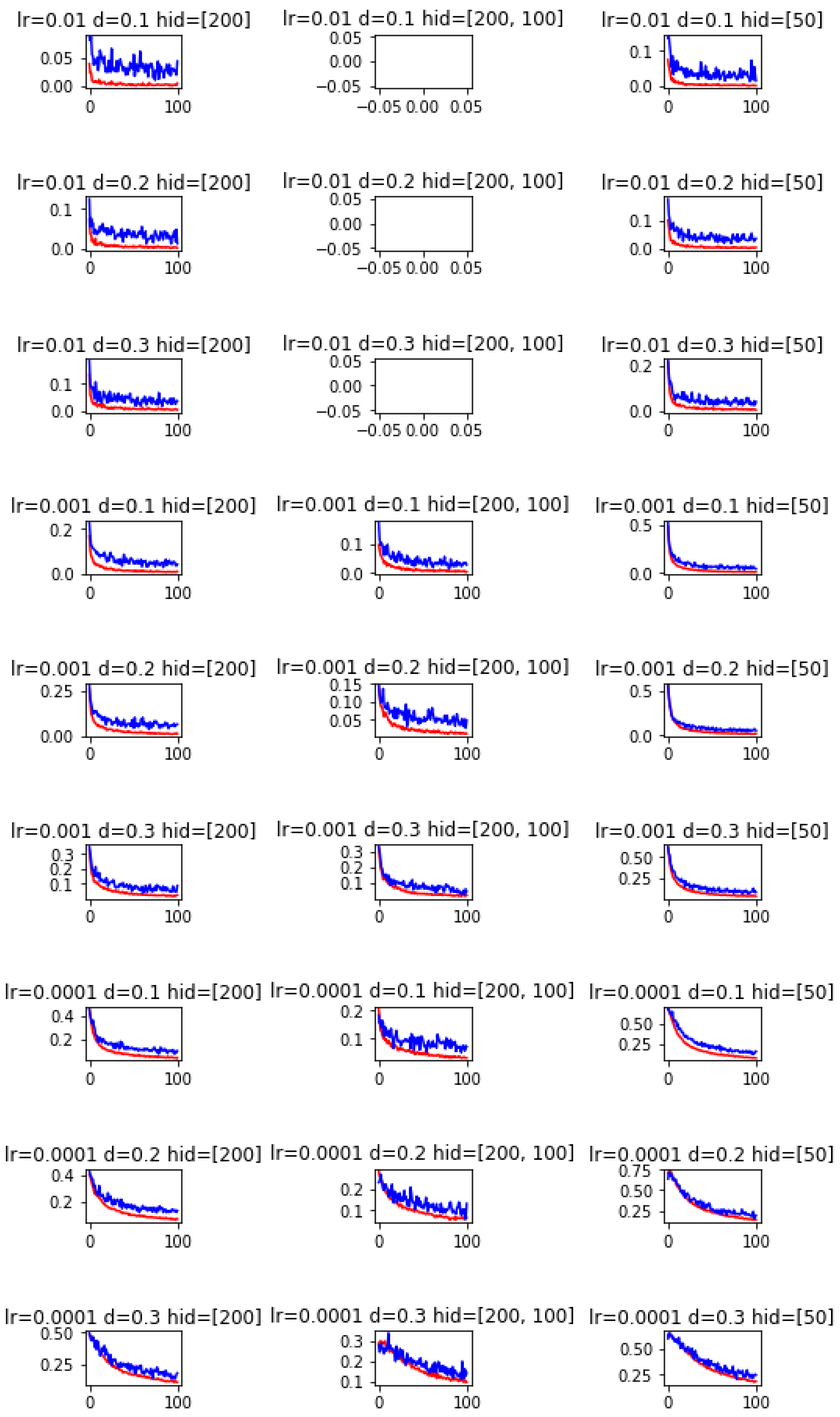
In [31]:

```
tune3_start = time.time()
lr = [0.01, 0.001, 0.0001]
dropout = [0.1, 0.2, 0.3]
hidden_dim = [[200], [200,100], [50]]
# create the table showing performance of each model. Here I use accuracy as the performanc
table_lr0 = PrettyTable(["lr = {}".format(lr[0]), "dropout = {}".format(dropout[0]), "dropo
            , "dropout = {}".format(dropout[2])])
table_lr1 = PrettyTable(["lr = {}".format(lr[1]), "dropout = {}".format(dropout[0]), "dropo
            , "dropout = {}".format(dropout[2])])
table_lr2 = PrettyTable(["lr = {}".format(lr[2]), "dropout = {}".format(dropout[0]), "dropo
            , "dropout = {}".format(dropout[2])])
acc_history = []

z=1

for i in range(len(lr)):
    for j in range(len(dropout)):
        for k in range(len(hidden_dim)):
            # train model with different parameters
            w_extended = network_weights(vocab_size=len(vocab),embedding_dim=w_glove.shape[
            hidden_dim=hidden_dim[k], num_classes=3)
            w_extended[0] = w_glove

            W, loss_tr, dev_loss = SGD(train_indices, train_label,
                                        w_extended,
                                        X_dev=development_indices,
                                        Y_dev=development_label,
                                        lr=lr[i],
                                        dropout=dropout[j],
                                        freeze_emb=True,
                                        tolerance=0.0000001,
                                        epochs=100, print_progress=False)
            # choose the best model by selecting the highest accuracy on the dev set
            preds_te = [np.argmax(forward_pass(x, W, dropout_rate=0.0)['prediction'])
            for x,y in zip(development_indices,development_label)]
            acc_history.append(accuracy_score(development_label,preds_te))
            plt.subplot(9,3,z)
            plt.plot(loss_tr, 'r')
            plt.plot(dev_loss, 'b')
            plt.title('lr={}'.format(lr[i])+' d={}'.format(dropout[j])+' hid={}'.format(hid

            z+=1

    if i == 0:
        table_lr0.add_row(["hidden_dim = {}".format(hidden_dim[0]), acc_history[i*9+0], acc
        table_lr0.add_row(["hidden_dim = {}".format(hidden_dim[1]), acc_history[i*9+1], acc
        table_lr0.add_row(["hidden_dim = {}".format(hidden_dim[2]), acc_history[i*9+2], acc
    if i == 1:
        table_lr1.add_row(["hidden_dim = {}".format(hidden_dim[0]), acc_history[i*9+0], acc
        table_lr1.add_row(["hidden_dim = {}".format(hidden_dim[1]), acc_history[i*9+1], acc
        table_lr1.add_row(["hidden_dim = {}".format(hidden_dim[2]), acc_history[i*9+2], acc
    if i == 2:
        table_lr2.add_row(["hidden_dim = {}".format(hidden_dim[0]), acc_history[i*9+0], acc
        table_lr2.add_row(["hidden_dim = {}".format(hidden_dim[1]), acc_history[i*9+1], acc
        table_lr2.add_row(["hidden_dim = {}".format(hidden_dim[2]), acc_history[i*9+2], acc

result = max(acc_history)
print('max accuracy:', result)
index_of_result = acc_history.index(result)
```

```
lr_index = (index_of_result)//9
dropout_index = (index_of_result%9)//3
hidden_dim_index = (index_of_result%3)
print('lr: ', lr[lr_index])
print('dropout: ', dropout[dropout_index])
print('hidden_dim: ', hidden_dim[hidden_dim_index])
print(table_lr0)
print(table_lr1)
print(table_lr2)
plt.subplots_adjust(left=4,
                    bottom=2,
                    right=5,
                    top=5,
                    wspace=2,
                    hspace=2)
plt.show()
tune3_end = time.time()
```

```
<ipython-input-14-d3c86f3461d1>:2: RuntimeWarning: overflow encountered in e
xp
  sig = np.exp(z)/sum(np.exp(z))
<ipython-input-14-d3c86f3461d1>:2: RuntimeWarning: invalid value encountered
in true_divide
  sig = np.exp(z)/sum(np.exp(z))
<ipython-input-14-d3c86f3461d1>:2: RuntimeWarning: overflow encountered in e
xp
  sig = np.exp(z)/sum(np.exp(z))
<ipython-input-14-d3c86f3461d1>:2: RuntimeWarning: invalid value encountered
in true_divide
  sig = np.exp(z)/sum(np.exp(z))
<ipython-input-14-d3c86f3461d1>:2: RuntimeWarning: overflow encountered in e
xp
  sig = np.exp(z)/sum(np.exp(z))
<ipython-input-14-d3c86f3461d1>:2: RuntimeWarning: invalid value encountered
in true_divide
  sig = np.exp(z)/sum(np.exp(z))
```

```
max accuracy: 0.9463087248322147
lr:  0.0001
dropout:  0.3
hidden_dim:  [50]
+--------------------------+--------------------+--------------------+----
-----------------+
|        lr = 0.01         |   dropout = 0.1    |   dropout = 0.2    |   d
ropout = 0.3     |
+--------------------------+--------------------+--------------------+----
-----------------+
|    hidden_dim = [200]    | 0.8926174496644296 | 0.912751677852349  | 0.9
194630872483222 |
| hidden_dim = [200, 100] | 0.3288590604026846 | 0.3288590604026846 | 0.3
288590604026846 |
|    hidden_dim = [50]     | 0.9194630872483222 | 0.9060402684563759 | 0.8
993288590604027 |
+--------------------------+--------------------+--------------------+----
-----------------+
+--------------------------+--------------------+--------------------+----
-----------------+
|        lr = 0.001        |   dropout = 0.1    |   dropout = 0.2    |   d
ropout = 0.3     |
+-------------------------+--------------------+--------------------+----
----------------+
|    hidden_dim = [200]   | 0.9194630872483222 | 0.8993288590604027 | 0.9
060402684563759 |
| hidden_dim = [200, 100] | 0.9060402684563759 | 0.9060402684563759 | 0.9
060402684563759 |
|    hidden_dim = [50]    | 0.9060402684563759 | 0.9194630872483222 | 0.9
060402684563759 |
+-------------------------+--------------------+--------------------+----
----------------+
+-------------------------+--------------------+--------------------+----
----------------+
|       lr = 0.0001       |   dropout = 0.1    |   dropout = 0.2    |   d
ropout = 0.3    |
+-------------------------+--------------------+--------------------+----
----------------+
|    hidden_dim = [200]   | 0.9060402684563759 | 0.9060402684563759 | 0.9
060402684563759 |
| hidden_dim = [200, 100] | 0.9060402684563759 | 0.9194630872483222 | 0.9
261744966442953 |
|    hidden_dim = [50]    | 0.8993288590604027 | 0.912751677852349  | 0.9
463087248322147 |
+-------------------------+--------------------+--------------------+----
----------------+
```

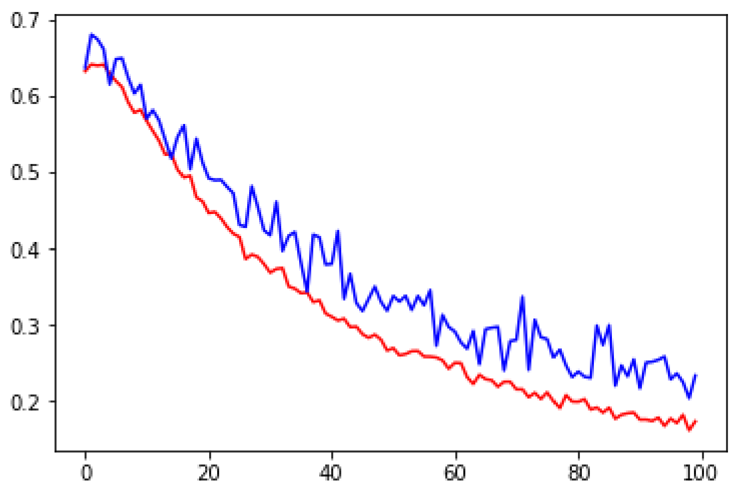

lr=0.01 d=0.1 hid=[200]
lr=0.01 d=0.1 hid=[200, 100]
lr=0.01 d=0.1 hid=[50]
lr=0.01 d=0.2 hid=[200]
lr=0.01 d=0.2 hid=[200, 100]
lr=0.01 d=0.2 hid=[50]
lr=0.01 d=0.3 hid=[200]
lr=0.01 d=0.3 hid=[200, 100]
lr=0.01 d=0.3 hid=[50]
lr=0.001 d=0.1 hid=[200]
lr=0.001 d=0.1 hid=[200, 100]
lr=0.001 d=0.1 hid=[50]
lr=0.001 d=0.2 hid=[200]
lr=0.001 d=0.2 hid=[200, 100]
lr=0.001 d=0.2 hid=[50]
lr=0.001 d=0.3 hid=[200]
lr=0.001 d=0.3 hid=[200, 100]
lr=0.001 d=0.3 hid=[50]
lr=0.0001 d=0.1 hid=[200]
lr=0.0001 d=0.1 hid=[200, 100]
lr=0.0001 d=0.1 hid=[50]
lr=0.0001 d=0.2 hid=[200]
lr=0.0001 d=0.2 hid=[200, 100]
lr=0.0001 d=0.2 hid=[50]
lr=0.0001 d=0.3 hid=[200]
lr=0.0001 d=0.3 hid=[200, 100]
lr=0.0001 d=0.3 hid=[50]

In [32]:

```
# After finding the best parameters, apply them to test data
w_extended = network_weights(vocab_size=len(vocab),embedding_dim=w_glove.shape[1],
                    hidden_dim=hidden_dim[hidden_dim_index], num_classes=3)
w_extended[0] = w_glove
W, loss_tr, dev_loss = SGD(train_indices, train_label,
                            w_extended,
                            X_dev=development_indices,
                            Y_dev=development_label,
                            lr=lr[lr_index],
                            dropout=dropout[dropout_index],
                            freeze_emb=False,
                            tolerance=0.0000001,
                            epochs=100)

# plot the graph of training loss and validation loss
plt.plot(loss_tr, 'r')
plt.plot(dev_loss, 'b')

# get the accuracy, precision, etc. of the best model
preds_te = [np.argmax(forward_pass(x, W, dropout_rate=0.0)['prediction'])
            for x,y in zip(test_indices,test_label)]

print('Accuracy:', accuracy_score(test_label,preds_te))
print('Precision:', precision_score(test_label,preds_te,average='macro'))
print('Recall:', recall_score(test_label,preds_te,average='macro'))
print('F1-Score:', f1_score(test_label,preds_te,average='macro'))
end_time = time.time()
time_used = (end_time-start_time-(tune1_end-tune1_start)-(tune2_end-tune2_start)-(tune3_end
print('It takes ', time_used, 'minutes to run the code')
```

```
train loss:  [0.6321546351664492, 0.6412462006056632, 0.6395999859830814, 0.
6409882058385525, 0.6290834082408635, 0.619385257413274, 0.6115224355598303,
0.5914428122343015, 0.5778758339183773, 0.5816800161245758, 0.56702410788150
51, 0.5538962770163545, 0.5411681695641615, 0.5230999292571602, 0.5238701552
44443, 0.5032638650276438, 0.4929738888459317, 0.49509209552089184, 0.466664
05040695236, 0.4615704999539093, 0.4461758037886564, 0.44769759304977386, 0.
4392241093486116, 0.4277507080817893, 0.41919282910803785, 0.414796591247032
33, 0.38650451360581456, 0.3923996832057615, 0.38923593843043824, 0.37978827
291138384, 0.3686315923609904, 0.37355100204671304, 0.37455316410147316, 0.3
503884687294774, 0.3476822809287633, 0.34167606151602725, 0.342738565245613
7, 0.32997593926412716, 0.3323620152538307, 0.3150102786356046, 0.3107820537
1784196, 0.30585956085895755, 0.30824408910099443, 0.2971888869391048, 0.297
776307778468, 0.28788771633180404, 0.2831617812943828, 0.2872640507938239,
0.28074645792973124, 0.26646985716461, 0.26962305753693067, 0.26012011595085
5, 0.2618418706960309, 0.2655844341729911, 0.26578584406260214, 0.2585309248
093595, 0.2583908125399167, 0.2573199064277416, 0.25354910480456694, 0.24259
405030596418, 0.2503117362060928, 0.24989540368447483, 0.2315363741073902,
0.2225565850137403, 0.23461610994656504, 0.22891888234701646, 0.227435103061
6666, 0.21896734406771162, 0.22540911074370962, 0.22549882479604466, 0.21554
88789451646, 0.21558020501433492, 0.20518221649493576, 0.21098021544422857,
0.20290313128662488, 0.21151516966045392, 0.20027318670016325, 0.19086451981
82557, 0.2079295831686074, 0.19949846498693138, 0.1992239558363409, 0.202678
7820719017, 0.1890309748294271, 0.19170092286271692, 0.18497908774030952, 0.
19149733494035806, 0.17694654544459815, 0.18214533263961868, 0.1841571578523
844, 0.18507023104014655, 0.17574384925462472, 0.175863804219746, 0.17403779
498387476, 0.17842593090832867, 0.16748510298215397, 0.17720197780141667, 0.
17129307165613036, 0.1820801773946572, 0.16172823156904775, 0.17348339283655
143]
```

```
------------------------------------------------
val loss:  [0.6367347385926706, 0.6804472291049181, 0.6743065363072879, 0.66
13800936225047, 0.6150572848245118, 0.6481359858582514, 0.6493354254354656,
0.6236517083610379, 0.6029635358445051, 0.6145494471004986, 0.56999228542678
1, 0.5812437817354491, 0.5679059259548123, 0.5426339886836669, 0.51783599105
62313, 0.545617926712454 2, 0.5612456883505865, 0.5036428344489672, 0.5438310
904011745, 0.512594881310272, 0.491790511255015, 0.4893226210310050 5, 0.4897
734923113793, 0.4804187967231071, 0.4721306452845319, 0.4306021292752333 4,
0.4280098947932459 7, 0.4814104972640458, 0.4536194636285514, 0.4234582781286
5405, 0.4169777598139469, 0.4609381207227977, 0.3970746675770359 7, 0.4161808
715573424, 0.4213171033779401 5, 0.3816699741676526 4, 0.343679149323445 4, 0.4
178942562247006 7, 0.4144013198873920 4, 0.3792957203128910 5, 0.38002130540688
75, 0.4226603158231373 4, 0.3342665625033962 6, 0.3671628441153939 6, 0.3287751
8102602454, 0.31822873973123167, 0.3343294948419882, 0.35033007130325733, 0.
33041092970050445, 0.31862906011202613, 0.33799526692758136, 0.3306248501150
3845, 0.33819722117704226, 0.3197691242097018, 0.3380617374487242, 0.3257758
400086867, 0.34560299250102955, 0.27269762733774433, 0.31316660058976953, 0.
2973622630331088, 0.2914151552758607, 0.27707846810107445, 0.268787043082780
9, 0.29212472700400444, 0.2484167502682485, 0.29455361465464047, 0.296286961
5638207, 0.29755870711116583, 0.24028771943294872, 0.27884102328695604, 0.28
03363155916619, 0.33714040970093595, 0.24098850380982398, 0.306824708063920
2, 0.28380688272848237, 0.2815527685968554, 0.25758270560161306, 0.267636390
39933856, 0.24672045587370856, 0.2316459642541296, 0.2388106683268504, 0.232
0687942585384, 0.23047211041397506, 0.2992288339517886, 0.27374339681296705,
0.29972416581199823, 0.22022308665561963, 0.2469312950597185, 0.232625990157
1779, 0.254849589099142, 0.21686564430822336, 0.2505863290964929, 0.25169108
83133812, 0.25420100129202744, 0.25853454383683294, 0.22856919164402473, 0.2
3637073003780248, 0.22503864492449513, 0.20387531768651934, 0.23366609527170
085]
Accuracy: 0.896551724137931
Precision: 0.8961146484636417
Recall: 0.8964957264957265
F1-Score: 0.8959742351046698
It takes  5.086473600069682 minutes to run the code
```



This model is neither overfitting nor underfitting. The only problem is that the validation loss curve shocks. When using smaller dropout rate, such as 0.1, the validation loss will become smoother. However, that model works worse than this model. It is kind of tradeoff, and I choose this model to be the best model.

Full Results

Add your final results here:

Model	Precision	Recall	F1-Score	Accuracy
Average Embedding	0.8527593830614076	0.8498253437383871	0.8494664856906962	0.8498331479421579
Average Embedding (Pre-trained)	0.8733180409890507	0.8720549981419546	0.8721743420278848	0.8720800889877642
Average Embedding (Pre-trained) + X hidden layers	0.8961146484636417	0.8964957264957265	0.8959742351046698	0.896551724137931

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

The best model is the model with hidden layer and the second best model is the model with pre-trained embedding layer. Pre-trained model is better than the first model is because pre-trained model use a trained embedding layer and this layer doesn't change while training the model. A trained embedding layer is reasonably better than the model with no pre-trained embedding layer. The model with hidden layer works better than the model without hidden layer. This is because the relu function in the hidden layer makes the classification formula generated by the model closer to the true classification formula. Thus, the third model works better than the second one and the second one works better than the first one.