[COM6513] Assignment: 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 (**2 marks**)
    - Perform a **Backward pass** to compute gradients and update all sets of weights (**3 marks**)
    - Implement and use **Dropout** after each hidden layer for regularisation (**1 marks**)
- Discuss how did you choose hyperparameters? You can tune the learning rate (hint: choose small values), embedding size {e.g. 50, 300, 500} and the dropout rate {e.g. 0.2, 0.5}. 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) 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? (**1 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 (**3 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. You must provide detailed explanations of your implementation, provide a detailed analysis of the results (e.g. why a model performs better than other models etc.) including error analyses (e.g. examples and discussion/analysis of missclasifications etc.) (**10 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 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.

Class 1: Politics, Class 2: Sports, Class 3: Economy

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. 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 **must** submit a Jupyter Notebook file (assignment_yourusername.ipynb) and an exported PDF version (you can do it from Jupyter: `File->Download as->PDF via Latex`, you need to have a Latex distribution installed e.g. MikTex or MacTex and pandoc). If you are unable to export the pdf via Latex, you can print the notebook web page to a pdf file from your browser (e.g. on Firefox: File->Print->Save to PDF).

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, 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 and discussion is as important as the implementation and accuracy of your models. Please be brief and consice in your discussion and analyses.

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, 26 Apr 2023** 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**, including Turnitin which helps detect plagiarism. Use of unfair means would result in getting a failing grade.

```
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).

```
train=pd.read_csv('data_topic/train.csv',header=None)
test=pd.read_csv('data_topic/test.csv',header=None)
dev=pd.read_csv('data_topic/dev.csv',header=None)
```

```
# Add: test data sample
test.sample(5)
```

	0	1
613	3	Charly Travers offers advice on withstanding t...
524	2	AP – Michael Schumacher clinched an unpreceden...
690	3	NEW YORK (Reuters) – U.S. blue chips were nea...
457	2	ATHENS (Reuters) – Ricardo Santos and Emanuel...
85	1	Fatah, the mainstream Palestinian movement, ho...

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.

```
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.

```
import re

# generate feature of n-grams
def generate_ngrams(tokens, n):
    ngrams = [tokens[i:i + n] for i in range(len(tokens) - n + 1)]
    return [' '.join(ngram) for ngram in ngrams]

def extract_ngrams(x_raw, ngram_range=(1, 3), token_pattern=r'\b[A-Za-z][A-Za-z]+\b
                   stop_words=[], vocab=set()):
   # Use regular expressions to split words
    tokens = re.findall(token_pattern, x_raw.lower())

   # remove stop words
    tokens = [token for token in tokens if token not in stop_words]

    # Generate n-grams
    ngrams_list = []
    for n in range(ngram_range[0], ngram_range[1] + 1):
        ngrams_list.extend(generate_ngrams(tokens, n))

    # filter n-grams according to the given vocabulary
    if vocab:
        ngrams_list = [ngram for ngram in ngrams_list if ngram in vocab]

    return ngrams_list
```

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

```
from collections import Counter

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=[]):
    # Initialize document frequency counter and n-gram counter
    df = Counter()
    ngram_counts = Counter()

    # Extract n-grams for each document and update the counter
    for x_raw in X_raw:
        ngrams = extract_ngrams(x_raw, ngram_range=ngram_range, token_pattern=token
        unique_ngrams = set(ngrams)
        df.update(unique_ngrams)
        ngram_counts.update(ngrams)

    # keep n-grams that appear in at least min_df documents
    if min_df > 1:
        df = {ngram: count for ngram, count in df.items() if count >= min_df}
        ngram_counts = {ngram: count for ngram, count in ngram_counts.items() if ng

    # Keep the top keep_topN frequent n-grams
    if keep_topN > 0:
        top_ngrams = sorted(ngram_counts, key=ngram_counts.get, reverse=True)[:keep
        df = {ngram: df[ngram] for ngram in top_ngrams}
        ngram_counts = {ngram: ngram_counts[ngram] for ngram in top_ngrams}

    # Create glossary
    vocab = set(df.keys())

    return vocab, df, ngram_counts
```

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

```
ngram_range = (1, 1)
min_df = 1
```

```
keep_topN = 0  # keep unigrams which match the condition

# use get_vocab() function
vocab, df, ngram_counts = get_vocab(train[1], ngram_range=ngram_range, min_df=min_d

# print result to check
print("Vocabulary:", vocab)
print("Document Frequencies:", df)
print("Raw Frequencies:", ngram_counts)
```

```
    Vocabulary: {'ann', 'throwback', 'johnson', 'warriors', 'generation', 'overhea
    Document Frequencies: Counter({'reuters': 631, 'said': 432, 'tuesday': 413, 'w
    Raw Frequencies: Counter({'reuters': 694, 'said': 440, 'tuesday': 415, 'new':
```

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

```
# create dictionary ID to word
id_to_word = {i: word for i, word in enumerate(vocab)}

# create dictionary word to ID
word_to_id = {word: i for i, word in enumerate(vocab)}

# print result
print("ID to Word:", id_to_word)
print("Word to ID:", word_to_id)
```

```
    ID to Word: {0: 'ann', 1: 'throwback', 2: 'johnson', 3: 'warriors', 4: 'genera
    Word to ID: {'ann': 0, 'throwback': 1, 'johnson': 2, 'warriors': 3, 'generatic
```

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:

```
# Function to convert a list of unigrams into a list of vocabulary indices
train1=list(train[1])
test1=list(test[1])
dev1=list(dev[1])

def words_to_indices(doc_unigrams, word_to_id):
    return [word_to_id[word] for word in doc_unigrams if word in word_to_id]
```

Then convert them into lists of indices in the vocabulary:

```
# Convert documents in train, dev, and test sets into lists of vocabulary indices
X_train_indices = [words_to_indices(extract_ngrams(doc, ngram_range=(1, 1), stop_wo
X_dev_indices = [words_to_indices(extract_ngrams(doc, ngram_range=(1, 1), stop_word
X_test_indices = [words_to_indices(extract_ngrams(doc, ngram_range=(1, 1), stop_wor

# Print example
print("Example Document (Train Set):", train1[0])
print("Example Document as Unigrams (Train Set):", extract_ngrams(train1[0], ngram_
print("Example Document as Vocabulary Indices (Train Set):", X_train_indices[0])
```

```
    Example Document (Train Set): Reuters - Venezuelans turned out early\and in la
    Example Document as Unigrams (Train Set): ['reuters', 'venezuelans', 'turned',
    Example Document as Vocabulary Indices (Train Set): [2512, 3297, 83, 4131, 649
```

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

```
def d(t):
    for i in range(len(t)):
        t[i] -= 1
    return t
y_train=d(list(train[0]))
y_test=d(list(test[0]))
y_dev=d(list(dev[0]))
```

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.

```
def network_weights(vocab_size=1000, embedding_dim=300,
                    hidden_dim=[], num_classes=3, init_val=0.1):

    # Initialize the weight dictionary
    W = {}

    # Create the embedding layer weight matrix
    W[0] = np.random.uniform(-init_val, init_val, (vocab_size, embedding_dim)).asty

    # Create the hidden layer weight matrices
    layer_input_dim = embedding_dim
    for i, layer_output_dim in enumerate(hidden_dim):
        W[i+1] = np.random.uniform(-init_val, init_val, (layer_input_dim, layer_out
        layer_input_dim = layer_output_dim

    # Create the output layer weight matrix
    W[len(hidden_dim) + 1] = np.random.uniform(-init_val, init_val, (layer_input_di

    return W
```

```
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`)

```
def softmax(z):
    # Shift the input values to avoid numerical instability issues
    z_shifted = z - np.max(z)

    # Compute the exponentials
    exp_z = np.exp(z_shifted)

    # Compute the softmax
    sig = exp_z / np.sum(exp_z, axis=-1, keepdims=True)

    return sig
```

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

```
def categorical_loss(y, y_preds):
    # Get the probability of the true class
    true_class_prob = y_preds[y]

    # Compute the categorical cross entropy loss
    loss = -np.mean(np.log(true_class_prob))

    return loss
```

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

```
def relu(z):
    # Create a copy of the input array to avoid in-place modifications
    a = z.copy()

    # Apply the ReLU function element-wise
    a[a < 0] = 0

    return a

def relu_derivative(z):
    # Create a copy of the input array to avoid in-place modifications
    dz = z.copy()

    # Compute the derivative of the ReLU function element-wise
    dz[dz <= 0] = 0
    dz[dz > 0] = 1

    return dz
```

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

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

and returns:

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

```
def dropout_mask(size, dropout_rate):
    # Generate a random vector with values in the range [0, 1)
    random_vec = np.random.random(size)

    # Create the dropout vector by thresholding the random vector with the dropout
    dropout_vec = (random_vec > dropout_rate).astype(np.float32)

    return dropout_vec
```

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

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

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

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

and returns:

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

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

    h_vecs = []
    a_vecs = []
    dropout_vecs = []

    # Pass input through the network
    for i in range(len(W)):
        if i == 0:  # Embedding layer
            h = np.sum(W[0][x, :], axis=0)
        else:
            h = np.dot(a_vecs[-1], W[i])

        h_vecs.append(h)

        if i < len(W) - 1:  # Apply ReLU and dropout for hidden layers
            a = relu(h)
            dropout_vec = dropout_mask(a.shape[0], dropout_rate)
            a *= dropout_vec

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

    # Compute the output layer probabilities
    output_probs = softmax(h_vecs[-1])

    # Save values in the out_vals dictionary
    out_vals['h_vecs'] = h_vecs
    out_vals['a_vecs'] = a_vecs
    out_vals['dropout_vecs'] = dropout_vecs
    out_vals['y_preds'] = output_probs

    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.

```
def backward_pass(x, y, out_vals, W, learning_rate, freeze_emb=False):
    y_preds = out_vals['y_preds']
    h_vecs = out_vals['h_vecs']
    a_vecs = out_vals['a_vecs']
    dropout_vecs = out_vals['dropout_vecs']

    # Gradients on the output layer
    g_t = y_preds.copy()
    g_t[y] -= 1
    # Update the weights of the last layer
    W[len(W) - 1] -= learning_rate * np.outer(a_vecs[-1], g_t)

    # Backpropagate through hidden layers
    for i in range(len(W) - 2, 0, -1):
        g_t = np.dot(W[i + 1], g_t) * relu_derivative(h_vecs[i]) * dropout_vecs[i]
        W[len(W) - 1] -= learning_rate * np.outer(a_vecs[-1], g_t)

    # Update the weights of the first layer (if not frozen)
    if not freeze_emb:
        g_t = np.dot(W[1], g_t) * relu_derivative(h_vecs[0]) * dropout_vecs[0]
        for i, word_idx in enumerate(x):
            W[0][word_idx, :] -= learning_rate * g_t

    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

```
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 = []

    for epoch in range(epochs):
        # Shuffle training data
        indices = np.random.permutation(len(X_tr))
        X_tr = [X_tr[i] for i in indices]
        Y_tr = [Y_tr[i] for i in indices]

        # Train on each document
        for i, x in enumerate(X_tr):
            y_true = Y_tr[i]
            out_vals = forward_pass(x, W, dropout_rate=dropout)
            y_preds = out_vals['y_preds']
            W = backward_pass(x, y_true, out_vals, W, lr, freeze_emb)

        # Compute training loss
        train_losses = []
        for i, x in enumerate(X_tr):
            y_true = Y_tr[i]
            y_preds = forward_pass(x, W, dropout_rate=0)['y_preds']
            train_losses.append(categorical_loss(y_true, y_preds))
        avg_train_loss = np.mean(train_losses)
        training_loss_history.append(avg_train_loss)

        # Compute validation loss
        if len(X_dev) > 0 and len(Y_dev) > 0:
            val_losses = []
            for i, x in enumerate(X_dev):
                y_true = Y_dev[i]
                y_preds = forward_pass(x, W, dropout_rate=0)['y_preds']
                val_losses.append(categorical_loss(y_true, y_preds))
            avg_val_loss = np.mean(val_losses)
            validation_loss_history.append(avg_val_loss)

            # Check for early stopping
            if epoch > 0 and validation_loss_history[-2] - validation_loss_history[
                break
        if print_progress:
            if len(X_dev) > 0 and len(Y_dev) > 0:
                print(f"Epoch: {epoch+1}/{epochs} | Training Loss: {avg_train_loss:
            else:
                print(f"Epoch: {epoch+1}/{epochs} | Training Loss: {avg_train_loss:

    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:

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

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

W, loss_tr, dev_loss = SGD(X_train_indices, y_train,
                            W,
                            X_dev=X_dev_indices,
                            Y_dev=y_dev,
                            lr=0.001,
                            dropout=0.2,
                            freeze_emb=False,
                            tolerance=0.01,
                            epochs=100)
```

```
Shape W0 (8931, 300)
Shape W1 (300, 3)
Epoch: 1/100 | Training Loss: 1.0159 | Validation Loss: 1.0395
Epoch: 2/100 | Training Loss: 0.9323 | Validation Loss: 0.9931
Epoch: 3/100 | Training Loss: 0.8478 | Validation Loss: 0.9410
Epoch: 4/100 | Training Loss: 0.7650 | Validation Loss: 0.8879
Epoch: 5/100 | Training Loss: 0.6870 | Validation Loss: 0.8321
Epoch: 6/100 | Training Loss: 0.6156 | Validation Loss: 0.7753
Epoch: 7/100 | Training Loss: 0.5532 | Validation Loss: 0.7242
Epoch: 8/100 | Training Loss: 0.4993 | Validation Loss: 0.6771
Epoch: 9/100 | Training Loss: 0.4523 | Validation Loss: 0.6402
Epoch: 10/100 | Training Loss: 0.4116 | Validation Loss: 0.6026
Epoch: 11/100 | Training Loss: 0.3759 | Validation Loss: 0.5712
Epoch: 12/100 | Training Loss: 0.3441 | Validation Loss: 0.5460
Epoch: 13/100 | Training Loss: 0.3160 | Validation Loss: 0.5215
Epoch: 14/100 | Training Loss: 0.2909 | Validation Loss: 0.5019
Epoch: 15/100 | Training Loss: 0.2677 | Validation Loss: 0.4850
Epoch: 16/100 | Training Loss: 0.2468 | Validation Loss: 0.4705
Epoch: 17/100 | Training Loss: 0.2283 | Validation Loss: 0.4556
Epoch: 18/100 | Training Loss: 0.2112 | Validation Loss: 0.4428
```

Plot the learning process:

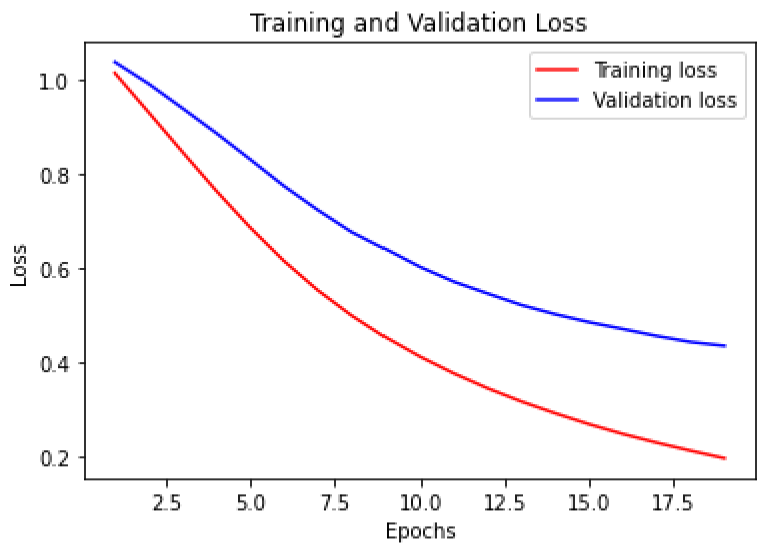
```
import matplotlib.pyplot as plt
```

```
def plot_learning_process(training_loss_history, validation_loss_history):
    epochs = range(1, len(training_loss_history) + 1)

    plt.plot(epochs, training_loss_history, 'r', label='Training loss')
    plt.plot(epochs, validation_loss_history, 'b', label='Validation loss')
    plt.title('Training and Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()

    plt.show()

plot_learning_process(loss_tr, dev_loss)
```



Compute accuracy, precision, recall and F1-Score:

```
preds_te = [np.argmax(forward_pass(x, W, dropout_rate=0.0)['y_preds'])
            for x,y in zip(X_test_indices,y_test)]

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

```
Accuracy: 0.8322222222222222
Precision: 0.8393300708115522
Recall: 0.8322222222222222
F1-Score: 0.831591553878873
```

Discuss how did you choose model hyperparameters ?

Choosing the model hyperparameters is an essential step in training a neural network. Ideally, you want to find a set of hyperparameters that results in the best performance on the validation set. There are several techniques to choose the hyperparameters, and I'll outline a few of them below:

Grid search: You can start by defining a range of values for each hyperparameter and perform an exhaustive search over all possible combinations. This method can be computationally expensive, especially when there are many hyperparameters or their value ranges are large.

Random search: Instead of searching exhaustively over all possible combinations, you can randomly sample a set of hyperparameter values and train the model with these values. This method is more efficient than grid search and often yields comparable results.

Bayesian optimization: This method models the unknown objective function (e.g., validation loss) as a probability distribution and iteratively updates the distribution based on the observed data. It balances the exploration and exploitation trade-off, allowing you to find the best hyperparameters more efficiently.

When choosing hyperparameters, you should consider:

Learning rate: The learning rate is critical because it determines how much the model's weights are updated during each step of gradient descent. A too large learning rate may cause the model to overshoot the optimal solution, while a too small learning rate may result in slow convergence. You can try various learning rates on a logarithmic scale, e.g., 0.1, 0.01, 0.001, and so on.

Dropout rate: The dropout rate is a regularization technique that helps prevent overfitting. You can try different dropout rates between 0 and 1 (e.g., 0.1, 0.2, 0.5) to find the best value that prevents overfitting while maintaining good performance.

Number of hidden layers and their sizes: The architecture of the network, including the number of hidden layers and the number of units in each layer, can significantly impact the model's performance. You can experiment with different architectures (e.g., 1 or 2 hidden layers) and varying numbers of units in each layer (e.g., 50, 100, 200).

Batch size: The batch size can impact both the training speed and the model's performance. Larger batch sizes can result in faster training but may lead to worse generalization. You can try different batch sizes (e.g., 32, 64, 128) to find the best trade-off between training speed and performance.

Number of epochs: The number of epochs determines how many times the model goes through the entire dataset during training. You can set a large number of epochs and use early stopping based on the validation loss to prevent overfitting. The tolerance parameter can be used to stop training if the improvement in validation loss is smaller than the specified threshold.

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.

```
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
```

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

```
def backward_pass(x, y, out_vals, W, learning_rate, freeze_emb=True):
    y_preds = out_vals['y_preds']
    a_vecs = out_vals['a_vecs']

    # Gradients on the output layer
    g_t = y_preds.copy()
    g_t[y] -= 1

    # Update the weights of the last layer
    W[len(W) - 1] -= learning_rate * np.outer(a_vecs[-1], g_t)

    # Backpropagate through hidden layers
    for i in range(len(W) - 2, 0, -1):
        g_t = np.dot(W[i], g_t) * (1 - a_vecs[i] ** 2)

        # Update the weights of the current layer
        W[i] -= learning_rate * np.outer(a_vecs[i - 1], g_t)

    # Update the weights of the first layer (if not frozen)
    if not freeze_emb:
        for word_idx in x:
            W[0][word_idx, :] -= learning_rate * g_t

    return W
```

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:

```
W[0] = w_glove
W, loss_tr, dev_loss = SGD(X_train_indices, y_train,
                           W,
                           X_dev=X_dev_indices,
                           Y_dev=y_dev,
                           lr=0.001,
                           dropout=0.2,
                           freeze_emb=True,
                           tolerance=0.01,
                           epochs=100)
```

```
    Epoch: 1/100 | Training Loss: 0.7848 | Validation Loss: 0.5632
    Epoch: 2/100 | Training Loss: 0.6986 | Validation Loss: 0.3769
```

```
preds_te = [np.argmax(forward_pass(x, W, dropout_rate=0.0)['y_preds'])
            for x,y in zip(X_test_indices,y_test)]

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

```
    Accuracy: 0.86
    Precision: 0.8663967882234861
    Recall: 0.86
    F1-Score: 0.85869250016039
```

Discuss how did you choose model hyperparameters ?

When choosing the model hyperparameters, there are several factors to consider. One common approach is to perform a search over a range of possible hyperparameter values and choose the combination that results in the best performance on the validation set. This can be done using grid search, random search, or more advanced techniques like Bayesian optimization.

Here's a list of hyperparameters you might consider tuning and some general guidelines for choosing their values:

Learning rate (lr): The learning rate is a crucial hyperparameter, as it determines the step size taken during weight updates. Typical values to consider are between 1e-5 and 1. It's common to start with a value around 0.001 and try different values on a logarithmic scale (e.g., 0.0001, 0.001, 0.01, 0.1).

Dropout rate: Dropout is a regularization technique that helps prevent overfitting. The dropout rate determines the proportion of neurons that are randomly "dropped out" or deactivated during training. Typical values to consider are between 0 and 1, with 0.5 being a common starting point. You can try values like 0.1, 0.2, 0.5, and 0.8.

Number of epochs: The number of epochs represents the number of complete passes through the training dataset. It's important to strike a balance between too few epochs, which might result in underfitting, and too many epochs, which could lead to overfitting. It's common to use early stopping with a patience (tolerance) value, which stops the training process when the validation loss does not improve for a certain number of consecutive epochs.

Hidden layer size: The size of the hidden layers in the network can affect the model's capacity to learn complex patterns. A larger hidden layer size may increase the model's expressive power but can also lead to overfitting. You can try different sizes like 50, 100, 200, or 300, depending on the complexity of the problem.

Embedding size: If you are not using pre-trained embeddings, you can also tune the size of the word embeddings in your model. Similar to hidden layer size, larger embedding sizes can capture more information but may also lead to overfitting. Typical values to consider are 50, 100, 200, or 300.

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?

```
def tanh_derivative(x):
    return 1 - np.tanh(x)**2
def backward_pass(x, y, out_vals, W, learning_rate, freeze_emb):
    y_preds = out_vals['y_preds']
    h_vecs = out_vals['h_vecs']
    a_vecs = out_vals['a_vecs']

    # Gradients on the output layer
    g_t = y_preds.copy()
    g_t[y] -= 1

    # Update the weights of the last layer
    W[-1] -= learning_rate * np.outer(a_vecs[-1], g_t)

    # Backpropagate through hidden layers
    for i in range(len(W) - 2, 0, -1):
        g_t = np.dot(g_t, W[i + 1].T)

        # Update the weights of the hidden layers
        W[i] -= learning_rate * np.outer(a_vecs[i - 1], g_t)

    # Update the weights of the first layer (if not frozen)
    if not freeze_emb:
        for word_idx in x:
            W[0][word_idx, :] -= learning_rate * np.dot(g_t, W[1].T)

    return W
```

```
hidden_layer_size1 = 100
hidden_layer_size2 = 50

W = [
    w_glove,  # Embedding layer
    np.random.randn(300, hidden_layer_size1) * 0.01,  # First hidden layer
    np.random.randn(hidden_layer_size1, hidden_layer_size2) * 0.01,  # Second hidde
    np.random.randn(hidden_layer_size2, 3) * 0.01  # Output layer
]

# Train and evaluate the deeper model
W, loss_tr, dev_loss = SGD(X_train_indices, y_train,
                          W,
                          X_dev=X_dev_indices,
                          Y_dev=y_dev,
                          lr=0.001,
                          dropout=0.2,
                          freeze_emb=True,
                          tolerance=0.01,
                          epochs=100)
```

```
Epoch: 1/100 | Training Loss: 1.0962 | Validation Loss: 1.0965
Epoch: 2/100 | Training Loss: 1.0770 | Validation Loss: 1.0818
Epoch: 3/100 | Training Loss: 0.9568 | Validation Loss: 1.0104
Epoch: 4/100 | Training Loss: 0.5273 | Validation Loss: 0.4487
Epoch: 5/100 | Training Loss: 0.3931 | Validation Loss: 0.2278
```

```
preds_te = [np.argmax(forward_pass(x, W, dropout_rate=0.0)['y_preds'])
            for x,y in zip(X_test_indices,y_test)]

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

```
Accuracy: 0.8888888888888888
Precision: 0.8896440235689159
Recall: 0.8888888888888888
F1-Score: 0.8871945266198953
```

Discuss how did you choose model hyperparameters ?

When choosing the model hyperparameters, there are several factors to consider. One common approach is to perform a search over a range of possible hyperparameter values and choose the combination that results in the best performance on the validation set. This can be done using grid search, random search, or more advanced techniques like Bayesian optimization.

Here's a list of hyperparameters you might consider tuning and some general guidelines for choosing their values:

Learning rate (lr): The learning rate is a crucial hyperparameter, as it determines the step size taken during weight updates. Typical values to consider are between 1e-5 and 1. It's common to start with a value around 0.001 and try different values on a logarithmic scale (e.g., 0.0001, 0.001, 0.01, 0.1).

Dropout rate: Dropout is a regularization technique that helps prevent overfitting. The dropout rate determines the proportion of neurons that are randomly "dropped out" or deactivated during training. Typical values to consider are between 0 and 1, with 0.5 being a common starting point. You can try values like 0.1, 0.2, 0.5, and 0.8.

Number of epochs: The number of epochs represents the number of complete passes through the training dataset. It's important to strike a balance between too few epochs, which might result in underfitting, and too many epochs, which could lead to overfitting. It's common to use early stopping with a patience (tolerance) value, which stops the training process when the validation loss does not improve for a certain number of consecutive epochs.

Hidden layer size: The size of the hidden layers in the network can affect the model's capacity to learn complex patterns. A larger hidden layer size may increase the model's expressive power but can also lead to overfitting. You can try different sizes like 50, 100, 200, or 300, depending on the complexity of the problem.

Embedding size: If you are not using pre-trained embeddings, you can also tune the size of the word embeddings in your model. Similar to hidden layer size, larger embedding sizes can capture more information but may also lead to overfitting. Typical values to consider are 50, 100, 200, or 300.

Full Results

Add your final results here:

Model	Precision	Recall	F1-Score	Accuracy
Average Embedding	0.839	0.832	0.832	0.832
Average Embedding (Pre-trained)	0.866	0.86	0.859	0.86
Average Embedding (Pre-trained) + X hidden layers	0.890	0.889	0.887	0.889

Please discuss why your best performing model is better than the rest and provide a bried error analaysis.

The best performing model is the one using Average Embedding with Pre-trained (GloVe) embeddings and additional hidden layers. The improved performance can be attributed to the following reasons:

Pre-trained Embeddings: The pre-trained GloVe embeddings have been trained on a large corpus and are capable of capturing semantic relationships between words better than randomly initialized embeddings. Using these pre-trained embeddings helps the model learn a better representation of the input data, which in turn improves its performance on the classification task.

Additional Hidden Layers: The deeper architecture allows the model to learn more complex and abstract features from the input data. The extra hidden layers help the model to better capture non-linear relationships between input features and the target classes. This results in improved performance on the classification task.

Error analysis:

Even though the best performing model shows improved performance, there might still be some errors in the predictions. Some possible sources of errors include:

Insufficient training data: The model may not have seen enough examples of certain classes or specific relationships between words during training. This might result in misclassifications for certain types of input.

Ambiguity: Some sentences might be inherently ambiguous or have multiple valid interpretations. In such cases, even a well-trained model might struggle to make the correct classification.

Hyperparameter choices: The choice of hyperparameters, such as learning rate, dropout rate, and the number of hidden layers, may not be optimal. There may be better choices that could further improve the model's performance.

To further improve the performance of the model, one could explore different architectures, use more training data, fine-tune the pre-trained embeddings, or experiment with different hyperparameters. Additionally, incorporating techniques such as regularization or batch normalization could help reduce overfitting and improve generalization to unseen data.